

TypeCalc: A Spreadsheet Implementation in TypeScript

Software Design Thesis
Course code: KISPECI1SE

August Hans Seierøe Pedersen (aupe@itu.dk),
Jacob Dinh Juulsgaard Breum (jadb@itu.dk),
Peter Klarskov Døssing (pekd@itu.dk),
Philip Boesen Blomholt (phbl@itu.dk)

Supervisor: Peter Sestoft (sestoft@itu.dk)

Github repository
<https://github.com/phbl-dev/TypeCalc>

June 2, 2025

Abstract

In this thesis, we detail the implementation of the novel TypeScript spreadsheet application TypeCalc. TypeCalc is based on previous research on spreadsheet implementation technology, and integrates this knowledge into an application that is operated via a browser interface and capable of representing tens of billions of cells in a row-column layout by utilising a virtualised grid, the specialised QT4 data structure, and efficient recalculation. The thesis describes how we used this foundation to implement a full-stack spreadsheet application with a back-end solely written in TypeScript and a front-end utilising React, and the particulars of the development process of both of these sides of the implementation. Specifically, we have highlighted the hierarchies and structures of cells, sheets and workbooks, the parsing of cell contents, the use of the QT4 data structure for sheet representation, and the tracking of cell dependencies and how to utilise these structures for efficient recalculation. Additionally, the thesis elaborates on our use of testing and documentation, the roles of these tools in software development, and especially in anticipation of the release of TypeCalc as open-source software for other researchers and developers to use and improve. Finally, we reflect on the process of developing spreadsheets, what facets were either easier or more difficult to implement than anticipated, where we deviated from our source material and the consequences of this, the use of TypeScript for spreadsheet implementation, the use of React, and the value of utilising third-party libraries.

Contents

1	Introduction	6
2	Background	7
2.1	TypeScript	7
2.2	Spreadsheet Terminology	8
2.2.1	What is a Spreadsheet?	8
2.2.2	Formulas, Functions, and Expressions	10
2.3	Spreadsheet Implementation Technology	11
2.4	Existing Spreadsheet Implementations in Typescript	12
3	Functionality and Project Goals	15
3.1	Our Contribution and Intended Learning Outcomes	15
3.2	Quality Goals	16
4	Methodology and Project Planning	17
4.1	Agile Development and Kanban	17
4.2	Pair Programming	19
4.3	Continuous Integration	20
4.4	Use of Artificial Intelligence	21
4.5	Repository Setup	21
5	Product Description	23
5.1	Cell Interaction	23
5.2	Sheet Selection	25
5.3	Header Functionality	26
5.4	Exporting and Importing files	26
5.5	Benchmark	26
6	Implementation	29
6.1	Overview of Implementation	29
6.2	API-Layer	29
6.2.1	Workbook Singleton Pattern	30
6.2.2	Cell Instantiation and Display	31
6.2.3	Workbook I/O	31
6.3	Documentation	34
7	Back-end	36
7.1	Overview of Back-end	36
7.2	Cells and SupportSets	41
7.3	Values	42

7.4	Expressions	43
7.4.1	Constant expressions	43
7.4.2	Reference expressions	43
7.4.3	Function expressions	44
7.5	Workbook and Sheets	44
7.5.1	QT4 Data structure	45
7.6	Cell Addressing and Referencing notations	48
7.7	Undo/Redo Functionality	50
7.7.1	Managing Undo/Redo History	50
7.7.2	Sheet.undo()	51
7.7.3	Sheet.redo()	52
7.8	Recalculation	53
8	Function Evaluation	55
8.1	Functions and Formula.js	55
8.2	New Function Type	55
8.3	FunCall.Make()	56
8.4	Expression Arrays	57
8.5	Handling Non-Strict Functions	57
8.6	FunCall.Eval()	58
9	Parsing	60
9.1	Parser Logic	60
9.2	Purpose and Library Considerations	60
9.3	SpreadsheetLexer	61
9.4	SpreadsheetParser	63
9.5	SpreadsheetVisitor	65
9.5.1	Integration with Codebase	66
9.6	TypeScript and Chevrotain	67
10	Front-end	68
10.1	Choice of Grid	68
10.2	Overview of Front-end Files	68
10.2.1	VirtualizedGrid Component	69
10.2.2	SheetHeader Component	72
10.2.3	SheetFooter Component	72
10.2.4	GridCell Component	72
10.3	Interaction Schemes & Front-end Features	73
10.3.1	Show Cell Supports and Dependents	74
10.3.2	Selecting Multiple Cells	75

11 Testing	77
11.1 Overview of Testing	77
11.2 Unit Testing	77
11.3 Component Testing	78
11.4 System Testing	78
12 Reflections	80
12.1 Reflections on the Product	80
12.1.1 Differences between TypeScript and C#	80
12.1.2 Cell reference expressions in formulas	81
12.1.3 Recalculation	82
12.1.4 Undo/Redo Functionality	83
12.1.5 React Integration	83
12.1.6 Shortcomings of react-window's Grid	84
12.1.7 On the Integration of Formula.js	85
12.1.8 Extension and Scaling of API-Layer	86
12.2 Reflections on the Process	87
12.2.1 On the Use of Agile	87
12.2.2 On the Value of Generative AI	88
12.2.3 Testing	89
12.2.4 Retrospective on Quality Goals	90
12.2.5 Learning Outcomes	92
12.3 Future Work	93
12.3.1 Notable Bugs	93
13 Conclusion	95

1 Introduction

In 1979, *VisiCalc* was released as the first commercially available spreadsheet application (Bricklin, 1999). Ever since then, the importance of such applications has only increased, and they have become a ubiquitous element of many industries, with Microsoft’s Excel and Google’s Sheets leading the charge (Duarte, 2025; Fallon, 2024). Spreadsheet programs offer a multitude of use cases and are used by millions of users daily to complete anything from analysing complex data to organising private budgets. Though modern applications afford many more features than *VisiCalc*, the core remains columns and rows of cells that can reference each other and perform various operations, allowing users to easily visualise data and relations within that data. In the words of *VisiCalc* co-creator Dan Bricklin: “The basic concept is [still] the same” (Bender, 2004, para. 13). This, of course, does not mean that spreadsheet programs have been perfected, and there is always room for improvement and new ideas. To this end, Peter Sestoft published his book *Spreadsheet Implementation Technology* (2014) in order to “enable others to make experiments with innovative spreadsheet functionality and with new ways to implement such functionality” (Sestoft, 2014b, p. ix). To complement this book, Sestoft has created the open-source spreadsheet implementation *CoreCalc*, written using Object-Oriented Programming in C#.

A major development since the release of both *Spreadsheet Implementation Technology* and *CoreCalc* is the emergence of TypeScript as one of the world’s most popular programming languages, enjoying varied use across all sectors of IT (JetBrains, 2024). Notably, TypeScript extends JavaScript, which is commonly used for creating websites as well as many mobile and desktop apps. TypeScript is, however, rarely used for spreadsheet implementations. While some commercial offers exist for spreadsheets, like the aforementioned Excel, and even some written in TypeScript, like Syncfusion Spreadsheet, we have yet to encounter a properly tested and documented open-source TypeScript spreadsheet implementation (Syncfusion, 2025). Based on *Spreadsheet Implementation Technology*, our project aims to help fill that gap with *TypeCalc*, an open-source browser-based spreadsheet implementation written in TypeScript and React, based on the concepts of *CoreCalc*. Throughout this report, we will describe how we designed and implemented *TypeCalc*, while at the same time describing relevant spreadsheet terminology and providing reflections on important design decisions and processes.

2 Background

Before addressing our implementation details, we describe the background of our project while at the same time describing our rationale for using TypeScript and important spreadsheet terminology.

2.1 TypeScript

An important decision in our project conceptualisation was the choice of TypeScript as the primary programming language. TypeScript is a "[...] strongly typed programming language that builds on JavaScript" (Microsoft, 2025c). The language emerged in 2012 as a response to the increasing size of JavaScript applications. As these projects grow, the lack of strong typing in JavaScript can lead to an enormous burden of work as bugs and inconsistencies become increasingly elusive in an application with hundreds of files. TypeScript eases this process because the language requires the programmer to declare the specific types of variables, return values, etc., in the code, such that the editor can catch errors at compile time. Instead of solving seemingly random and hard-to-trace bugs during run-time, TypeScript allows us to use modern programming tools, such as the static code analysis in developer environments, to identify the issues before they enter production (Microsoft, 2025c). The addition of these tools helps to ensure type safety, eases maintainability, and by extension scalability (Panchasara, 2021). However, TypeScript itself is never run or executed directly. Instead, TypeScript is transpiled to functionally equivalent JavaScript code, which is what we then run in the browser along with HTML and CSS (Microsoft, 2025c). What this means is that TypeScript is, in essence, a superset of all JavaScript functionality with an added strict type system to help developers.

The reasons we chose to use TypeScript are many. One is its relevance as a language in the contemporary software landscape. In the JetBrains 2024 State of the Developer Ecosystem released, TypeScript is highlighted as the most promising programming language in terms of its fast growth, adoption rate, and other similar metrics (JetBrains, 2024). Another is a consideration of the use case that we have. In the same paper, we can find a metric for the general use of TypeScript, and we can see that it is extremely popular for creating user interfaces and websites, both of which are relevant to our project. Since we want to create a browser-based graphical user interface (henceforth GUI) for our spreadsheet, this is an obvious choice when compared to the more desktop-native fit of other languages like C#. TypeScript may not be the best-performing language for large spreadsheets with many CPU-heavy computations, but the browser integration is phenomenal (Thomas & Steiner, 2024). Furthermore, Anders Hejlsberg, the lead engineer behind TypeScript, has recently announced significant performance changes are coming to the language as a result of changing to a Golang-based compiler (Hejlsberg, 2025). With access to libraries like React, we have been able to build a fast and scalable GUI for the browser in short fashion,

which we will elaborate on later. Many other languages have been used for spreadsheets. For example, Microsoft Excel, henceforth Excel, is written mostly in C++ (Antoun & Zaika, 2014), a language that is often associated with efficiency in terms of memory and speed. TypeScript’s memory usage is defined by dynamic allocation and garbage collection, and does not support the same kind of low-level control that C++ does (Mozilla, 2025c). Still, the language exhibits strengths in other areas and is extremely useful for developing browser and web-based applications. Google Sheets, a major competitor of Excel, used JavaScript for its calculation engine until recently (Thomas & Steiner, 2024).

Lastly, none of the group members had any experience with TypeScript prior to this project, and saw it as an opportunity to learn and grow as software professionals and academics, exploring an area of IT that is extremely relevant and useful, but has not found its way into our programme curricula. To this end, we set it as a primary project goal that we wanted to learn the ins and outs of development in TypeScript, its advantages and use cases, peculiarities and frustrations, and to gain new insights on the differences between developing full-stack programs for the browser as opposed to the smaller, more constrained projects we have done in the past as part of our education.

2.2 Spreadsheet Terminology

Going forward, we will be using terminology related to spreadsheet programs liberally. This pertains generally to the overall structure of spreadsheets, which consist of workbooks, worksheets (or commonly sheets), cells and formulas. A cell is the singular unit in a spreadsheet grid. To the user, it is a small rectangle with an address that contains a text field. The text field takes user-provided input, usually in the form of either constants (e.g. 123 or "Hello") or formulas (e.g. =10+10 or =SUM(10, 10)), and translates these inputs to different mathematical operations, references to other cells or sheets, and function calls (Sestoft, 2014b).

Together, all the cells in a single spreadsheet grid constitute a sheet of which there can be multiple. This collection of sheets together forms a workbook (Sestoft, 2014b). The sheets inside a workbook are related and can each reference one another in their cells. Apart from a collection of cells, a sheet has a name that is used for referencing. The conceptual model (Norman, 2013) is that of a binder of documents - the workbook is the binder, the sheets are the individual documents and the tables on them, and the cells are the entry spaces in the tables of these documents.

2.2.1 What is a Spreadsheet?

The definition of a spreadsheet is simultaneously broad and very specific - most will recognise applications like Excel and Google Sheets as spreadsheets, but the concrete definition is not found in such a comparison. In his 2014 book, Sestoft dedicates a 24-page chapter to

the exploration and explanation of the concept of spreadsheets (Sestoft, 2014b). Spreadsheets constitute programs that have a wide range of functionality and can perform specific operations. A spreadsheet is shown as a two-dimensional grid of columns and rows that constitute a sheet of individual cells. These cells have text fields wherein a user can type formulas or constants. One cell, called the *active cell*, is highlighted - usually either by a thicker border, different colour, or both, and is the cell that is currently being written into, whether the user decides to write directly in that cell or via the *formula box* at the top, as seen in Figure 1. The formula box is a text field that writes to the active cell and usually allows for different user interactions. For example, in Excel and in our application, TypeCalc, pressing an arrow key while in a cell will jump to the next cell in that direction, while pressing an arrow key in the formula box will move the cursor caret in the text.

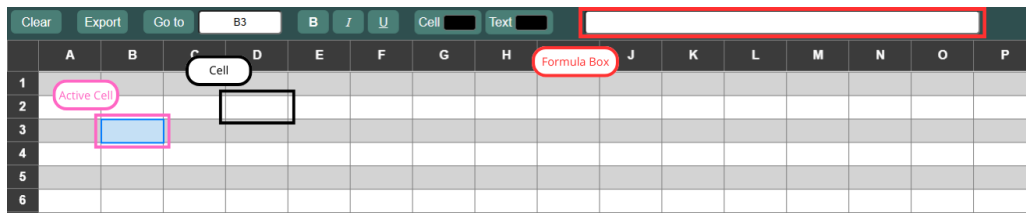


Figure 1: Snippet from TypeCalc showing a cell, the active cell and the formula box

Cells also have addresses, or references, that can be used to access their values from other cells. That is, a cell can depend on the value of another cell. A cell a is said to *directly depend on* another cell b if a 's value is determined in some way by the value of b (Sestoft, 2014b). Conversely, cell b is said to *directly support* cell a . The cells update and recalculate automatically when the values of the cells that they depend on change. This recalculation can determine cycles that would otherwise cause undefined behaviour, for example, when two cells both depend on each other, or when a cell refers to its own value. Along with the detection and propagation of other errors that reasonably occur in spreadsheets, such as the improper use of expressions for a function or syntax errors in cell input, the detection of dependency cycles is an important part of the recalculation in TypeCalc.

B1	A	B
1	=B1 + 1	=A1 + 1

Figure 2: Example from TypeCalc of a cycle that should result in an error value.

The purpose of recalculation is "[...] the automatic and consistent recalculation and re-display of results whenever the user has edited some data" (Sestoft, 2014b, p. 8). A

spreadsheet is used as a tool that dynamically updates during use. When a cell is updated, we expect its supported cells to update as well. Recalculation is responsible for this, and it is therefore a central mechanism that affects the efficiency and reliability of a spreadsheet (Sestoft, 2014b). Recalculation should maintain consistency in formulas and the cells that the formulas refer to. It should be efficient in terms of time and space, and should therefore not reevaluate unedited cells. Because of this, recalculation causes some algorithmic complications. There are several possible solutions to this problem and ways to create an efficient recalculation of cells, all of which have merits. In Section 7.8, we describe the recalculation methods used in TypeCalc.

A spreadsheet program also typically provides a library of functions such as `SUM()` and `LOG()`. Some of these can be *volatile*, meaning that the values are non-deterministic, like the function `RAND()` that returns a random number, and as such will produce a new output whenever the cell is recalculated. Additionally, some functions can be *non-strict*, meaning that not all of their arguments are necessarily evaluated at the time they are called. For example, the function call `=CHOOSE(A1, "Safe", F14)` may evaluate to two different values depending on the value of cell A1 (either 1 or 2), and the value of cell F14 will only be recalculated if necessary for this evaluation (i.e. the value of cell A1 is 2).

2.2.2 Formulas, Functions, and Expressions

Formulas, functions and expressions are three essential concepts in spreadsheet terminology, and they are easily confused with each other because of their connectedness. A formula consists of an expression together with the computed value of the expression (Sestoft, 2014b). It is written in a cell, starting with an equal sign, followed by the expression (see the red box in Figure 3). Thus, a formula is used to represent the contents of a cell when the cell contains an expression. The expression of the formula can consist of one or more expressions because expressions can be nested. An expression can be constants (e.g. numbers, text or booleans), function calls (e.g. `SUM(...)`), arithmetic operators (e.g. `*` and `+`), cell references (e.g. A1), and cell area references (e.g. A1:C3) (Sestoft, 2014b).

Spreadsheet functions can be of different types, such as mathematical functions (e.g. `ABS(...)` in Figure 3), logical functions (e.g. `IF(...)` which returns one value if a condition is true, and another if it is false), cell area functions (e.g. `MODEMULT(...)`, see Figure 4), and date and time functions (e.g. `TODAY()` which returns today's date) (Ghalimi et al., 2025a; Sestoft, 2014b). Function calls are a type of expression that calls these functions in formulas. Many spreadsheet functions take one or more arguments in the form of expressions, but some functions take zero arguments, such as `TODAY()` and `PI()`, which returns the value of pi.

As mentioned, functions can be strict or non-strict. Strict functions like `SUM(...)` eval-

uate all of their arguments, while non-strict functions such as `IF(...)` only evaluate the arguments they need (Sestoft, 2014b). Some functions evaluate to a single value, while other functions evaluate to multiple values, i.e. array-valued results (Sestoft, 2014b). Therefore, the array-valued result will be shown in multiple cells called array formulas. An example of a function that returns an array-valued result is `MODEMULT(...)`, which is seen in Figure 4.

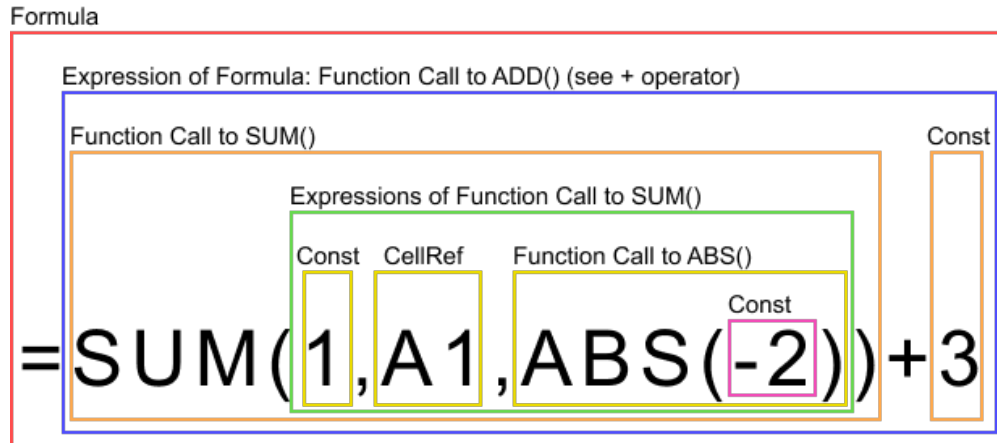


Figure 3: Illustration of how formulas, functions and expressions are related to each other. The red box surrounds a formula which is written in a spreadsheet cell. The blue box surrounds the expression of the formula, which in this case is a function call because the arithmetic operator `+` calls the function `ADD(...)` that adds two numbers. To the left of `+`, a function call to `SUM(...)` is seen in an orange box, which sums a range of values, and to the right, a number constant 3 is seen in the second orange box. The green box surrounds the expressions of the function call to `SUM(...)`, which are separated in smaller yellow boxes. The first yellow box from the left is a number constant 1, then the second yellow box contains a cell reference to the cell A1, and the third, a function call to `ABS(...)`, which returns the absolute value of a number. The pink box contains the expression given to `ABS(...)`, which is a number constant -2. If cell A1 evaluates to 10, this formula will evaluate to 16.

	A	B
1	=MODEMULT([1,2,2,3,3])	
2		

	A	B
1	2	
2	3	

Figure 4: The `MODEMULT` function returns an array-valued result containing the most frequent number(s) of an array or cell area. For example, `=MODEMULT([1,2,2,3,3])` to the left returns `[2,3]` to the right because 2 and 3 are both the most frequent numbers.

2.3 Spreadsheet Implementation Technology

Our project is based on the ideas, concepts and descriptions of the 2014 book, *Spreadsheet Implementation Technology (SIT)* by Peter Sestoft (Sestoft, 2014b). The book contains an overview of a design, as well as intricate details of the individual elements that constitute a spreadsheet program. From this source, we derive the main data structure, the QT4 simplified quadtree, for representing our sheet in the back-end. We will be referring to this data structure in the following sections and elaborate on it in Section 7.5.1. We

also follow the implementation details given for such spreadsheet-specific functionality as keeping track of supporting cells, standard minimal recalculation and cell referencing. Our implementation of all of these will be thoroughly explained in Section 7.

Another primary source is the practical implementation, CoreCalc, that Sestoft has created in relation to the book. CoreCalc is a desktop-based implementation of the concepts in SIT, written primarily in C#, and is available as open-source under an MIT license (Sestoft, 2014a). CoreCalc has been instrumental in our work, as we have often followed the implementation details that are accessible as part of the code. Frequently, we have made direct translations of the CoreCalc source code to TypeScript in order to exactly replicate functionality in the back-end. This has been exceptionally helpful and useful in our own work, although at times also problematic. Some functionality in the source code is very C# native and hard to translate directly. We will elaborate on this in both Section 7 and Section 12.

2.4 Existing Spreadsheet Implementations in Typescript

As part of scoping out our project, we conducted a thorough search of existing spreadsheet implementations. A lot of products exist in this space, including widespread and popular solutions, such as Excel and Google Sheets, to name a few. Common for all of these is the fact that they are proprietary software products, so we have no immediate way of knowing their implementation details. While we will be using these products throughout the report as baseline examples, they are not directly comparable to our open-source software.

There do exist a number of open-source spreadsheet solutions, however, which fall into two categories that determine their value to this project, based on the programming languages and tools that were used to develop them. For example, while LibreOffice and Gnumeric are well-known open-source spreadsheet application, they are written in C++ and C respectively and is thus not as interesting to our project (Duck, 2019; Goldberg, 2025).

In the other category, we find open-source spreadsheet software implemented in TypeScript. We have scoured various sources on the web, including papers that cite SIT, broad searches on Google Scholar and arXiv, and deep dives into GitHub repositories found via GitHub forks or in their own search engine. Amongst these, only our exploration of GitHub gave us notably valuable insight, as there simply was not much academic literature that was relevant to the field of spreadsheet implementation. On GitHub, we have found some promising solutions, but also many more unpolished, abandoned or otherwise unfinished projects. They are usually either the result of some educational activity and thus exploratory in nature, meaning that the author was trying to learn the basics for spreadsheet implementation and abandoned the project half-way, or the solutions are presented as functional pieces of software, that then prove to be incomplete and aborted upon further

inspection of the code. Examples of such projects can be found here, although we have reviewed several tens of these types of codebases:

- ExcelTS: <https://github.com/wx-chevalier/excel.ts> - Unfinished, partially written in Mandarin. Updated 5 years ago.
- ZXSheets: <https://github.com/jxlpzqc/ZXSheets> - Unfinished. Updated 3 years ago.
- JoselynDRF spreadsheet: <https://github.com/JoselynDRF/spreadsheet> - Unfinished. Updated 3 years ago.
- NodeSpreadsheet: <https://github.com/theoephraim/node-google-spreadsheet> - Is a Google Sheets wrapper. Updated recently.

Digging through these, we also come across a moderate swath of more finished products. These are all great solutions, among which we have tried to find our own niche in an unexplored area. We observed a number of great decisions in these solutions, some of which we decided to use ourselves. This includes the use of the Formula.js package for implementing Microsoft Excel functions in the spreadsheet (Ghalimi et al., 2025a), and the use of a virtualised grid to render visible cells in the viewport exclusively (Handsontable, 2025a). Conversely, the suboptimal documentation, data structure use and UIs of some of these solutions guided us in identifying a gap in the field and in our own pursuit of what meaningful improvements to a spreadsheet program can constitute. In summation, we do not find a solution that fulfils all the criteria we set in our project goals. Below is an overview table of these solutions.

Name	UI	Language	Tests	Sheet	Documentation	Link	Last Updated
vogtb	No	TypeScript	Yes	Hash Map	Mid	GitHub	2018
EtherCalc	Yes	LiveScript	Unknown	Hash Map	Yes, but hard to read	GitHub	2020
Handsontable	No	TypeScript	Yes	2D Array OR Hash Map (Dense vs Sparse)	Excellent	GitHub	2025
react-spreadsheet	Yes	TypeScript	Yes	2D Array	Excellent	GitHub	2025
TypeCalc	Yes	TypeScript	Yes	QT4	Excellent	GitHub	2025

Table 1: Overview of relevant open-source spreadsheet implementations

3 Functionality and Project Goals

In this section, we will introduce the goals of our project and our contribution. In doing so, we also present our intended learning outcomes for the process. Lastly, we provide an overview of the quality goals we created to measure TypeCalc against.

3.1 Our Contribution and Intended Learning Outcomes

Based on our literature review, we identified a gap that we could fill amongst all of the more finished spreadsheet implementations that we found. TypeCalc's niche is that it is a spreadsheet implementation that is written in TypeScript, open-source, has thorough documentation, has an easily navigated GUI, and employs a suitable and efficient data structure - in our case, the QT4 simplified quadtree. While there is some overlap with other spreadsheet implementations, none of the ones we found possessed all four of these characteristics. As alluded to in Section 2, we notably did not find anyone utilising the QT4 data structure, instead opting for more common data structures, such as arrays or hash maps. As such, early on in our project, after the literature review, we decided on the following points as our project's contribution to the field of spreadsheet implementation:

1. The implementation of a spreadsheet program in TypeScript, following the overarching guidelines of CoreCalc described in SIT (Sestoft, 2014b).
2. A browser-based user interface for easy interaction with the spreadsheet, following the column and row table visualisation.
3. A suite for testing functionality.
4. Thorough documentation of the code and its use.
5. The release of all the above as open-source code for further iteration, experimentation and/or research.

We also established the following list of learning outcomes for the project, as goals for what we personally wanted to learn throughout the process. That is, by the end of the project, we wanted to be able to:

1. Utilise HTML, basic CSS, and TypeScript, as well as UI principles to develop a dynamic application.
2. Describe the processes and challenges involved in implementing spreadsheet functionality.
3. Implement efficient data structures to manage spreadsheet data and optimise performance for large datasets.
4. Apply algorithms to handle formula evaluation, dependency tracking, and recalculation of cells efficiently.

5. Thoroughly document implemented functionality and design choices for an open-source project on GitHub.
6. Perform extensive testing to evaluate and document code quality by creating a test suite.
7. Organise and manage a medium-sized software development project by applying Agile principles and iterative development methodologies.

3.2 Quality Goals

To help serve as a guideline for our project, we established 20 quality goals for the product. These were based on findings from our literature review and research on existing spreadsheet implementations, particularly in regards to these implementations' features. Given the constraints of scope and time for a thesis project, we were naturally never going to be able to replicate, e.g., Excel or Google Sheets. Therefore, we would instead use the quality goals to evaluate the success of TypeCalc. Our expectation was to accomplish the first 13 goals at a minimum, with the remaining 7 being ones we could strive towards. Our quality goals were as follows:

1. Develop a simple prototype of a spreadsheet with a grid size of 20x20.
2. Implement basic functions such as ADD, SUM and MINUS.
3. Implement cell formatting features (e.g., bold, italic, background colour).
4. Implement the prototype in a web browser.
5. Make an "import file" function that allows users to import a spreadsheet file.
6. Implement undo/redo functionality for spreadsheet operations.
7. Increase the grid size to 2,000x4,000.
8. Implement a suite for testing of functionality.
9. Implement advanced functions such as FREQUENCY and MODEMULT.
10. Allow the sheet to show cell dependencies and dependents.
11. Automatic recalculations of cell operations
12. Make the spreadsheet implementation portable over a range of different web browsers.
13. Implement algorithms and data structures to handle larger spreadsheets.
14. Increase the grid size to 16,000x1,000,000 or more.
15. Implement an "export file" function for a common spreadsheet format.
16. Allow for multiple sheets that can reference each other.
17. Implement autofill in the spreadsheet.
18. Make the cells draggable.
19. Implement visual diagrams such as single-line diagrams and histograms.
20. Allow users to collaborate in the spreadsheet.

We will return to the quality goals and reflect on them later in Section 12.

4 Methodology and Project Planning

In this section, we cover our use of Agile as a software development project planning tool, highlighting our use of the Agile approach, Kanban, we describe our use of Agile practices, such as pair programming and continuous integration, and finally we cover our GitHub repository setup and our use of artificial intelligence tools in this project.

4.1 Agile Development and Kanban

To structure this project, we have made use of Agile; a commonly used software development approach to larger team-based projects. Agile saw its rise during the 1990s as a reaction to existing software development practices, such as the waterfall model (Sommerville, 2016). These practices were criticised as being too rigid and bureaucratic, often using more time planning the design of the software compared to designing the software itself (Sommerville, 2016). In addition to this, the rigidity of these models often made a change in requirements a costly and time-consuming endeavour. Agile presented itself in stark contrast to this, emphasising an iterative approach to software development, where a large project is broken into smaller sprints that can be completed faster in increments. There are a variety of benefits to this way of working; for one, software can be shipped faster, since parts of it are being built at a time, showcasing value for the stakeholder. This makes it easier for stakeholders to be involved in the system design process, and in turn, easier to adjust requirements for the software system. By working iteratively, software development teams are able to work more efficiently, and can plan their working process in a more flexible manner with respect to the amount of workload they have (Sommerville, 2016). Today, Agile has more or less become an umbrella term for Agile development frameworks, such as XP, Scrum, Kanban, etc. These frameworks are all heavily inspired by the core tenets of Agile development, which emphasise "Individuals and interactions over processes and tools; Working software over comprehensive documentation; Customer collaboration over contract negotiation; and responding to change over following a plan." (Beck et al., 2001)

In our project, we have made use of the Agile project management framework *Kanban*, which focuses on the visualisation of tasks to manage workflows. It emphasises the notion of transparency, capacity and continuous improvement through the use of Kanban boards, where tasks are being transferred between different stages to signify their progress (Asana, 2025). In Kanban, a large project can be broken down into a series of tasks, which are added to a backlog. Tasks are added onto a pending column in the Kanban board, signalling that a task can be "pulled" by a team member. As tasks are being completed, new backlog tasks can be added onto the Kanban board (Asana, 2025). A Kanban board is generally characterised by a board containing columns: *backlog*, *pending*, *in progress*, *ready for review*, and a *done* section. These columns help the team keep track of who does what

and what needs to be done. With Kanban project planning, the goal is to progress through tasks as fast as possible, such that new tasks can be assigned. Assigning too many tasks at once can slow down the workflow for other parts of the software development process, since it may cause the team to work inefficiently with many tasks, instead of efficiently on a few. Additionally, certain tasks may require others to be finished, which can slow down the process if a team member is over encumbered by tasks (Asana, 2025). We chose to use Kanban as our project management framework, since it is easy to set up and integrate into the existing team practice we had. Furthermore, the Kanban approach is one of the more forgiving frameworks for inexperienced Agile users.

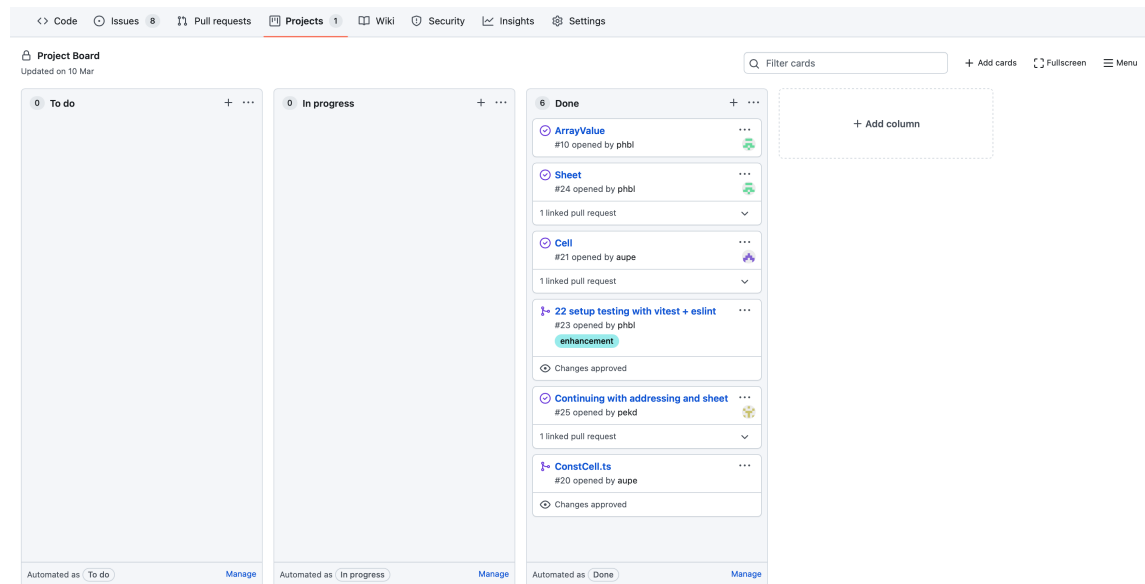


Figure 5: Our initial issue planning approach

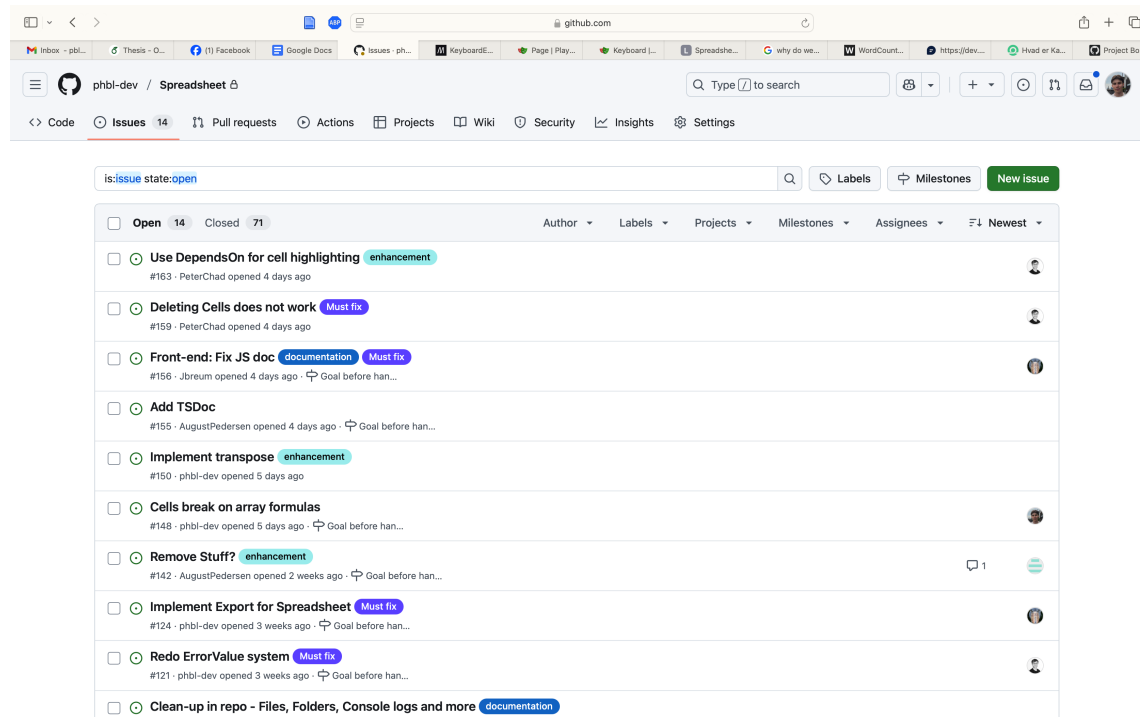


Figure 6: Our final and functioning planning approach

Our initial Kanban board was created using Trello, a Kanban board management tool (Atlassian, 2025). We would later move on to use *GitHub projects*, which offered a better integration with our repository setup. Our Kanban board was done in a somewhat simple manner, and contained three columns: *Todo*, *in progress*, and *done* as seen in Figure 5. This allowed for a simple overview of the current tasks and those responsible for handling them. Initially, our tasks were created vaguely, but throughout the project evolved to be more specific as more knowledge was gathered. Our process of assigning tasks to team members was primarily done at the beginning of each week. Here we spent time reflecting on the previously assigned tasks, addressing potential concerns, and planning for unexpected difficulties of certain tasks. Later on, we adapted our use of the Kanban board to GitHub Issues as seen in Figure 6, which will be addressed in Section 12.2.1.

4.2 Pair Programming

While some tasks were relatively straightforward and could easily be solved by a single team member, others were more logically challenging. In these scenarios, we made use of the Agile development practice *pair programming*. Pair programming is the practice of two people actively working together on a software issue, where one person writes the code, and the other is in charge of defining what should be written (Sommerville, 2016). The idea is that the pair is in charge of the software together, and being a pair provides support that in the end can be essential in solving the issue (Sommerville, 2016). The notion of

pair programming is useful on several fronts; it adds a sense of collective ownership to a singular problem, increasing active participation from both participants. Furthermore, it also increases the chance of various errors being discovered earlier, and helps with early refactorisation of code, saving time spent on refactoring and debugging later (Sommerville, 2016). In our project, we have found pair programming to be a particularly helpful practice, since our project has both a larger back-end and front-end section. To best coordinate our time, we used pair programming in scenarios where one person might have more knowledge regarding one part of the code, while the partner was more keen on another part, which had to be integrated.

4.3 Continuous Integration

Continuous integration, often abbreviated CI, is a software development practice originating within the DevOps community. In essence, it is the practice of running automated tests and/or scripts on new commits to ensure that new changes to the codebase are merged, such that they are in a working state afterwards (Gift et al., 2019). It sets a working standard for what the system should be able to do, while allowing developers to more freely work and integrate changes with the main branch as long as they do not break the code (Gift et al., 2019). In our project, we have made use of GitHub Actions; a part of GitHub which offers many different options for automating processes within the GitHub environment, such as running pre-merge analysis and post-commit actions. In addition to this, GitHub Actions offers a robust selection of open-source solutions, significantly reducing the amount of boilerplate code needed for automation scripts (Github, 2025a).

Our CI chain combines both pre-commit and post-push actions. In other words, a series of actions are performed before each commit locally, ensuring that code formatting is done correctly using the code formatting tool *prettier* (Prettier, 2025), and the documentation is updated and pushed alongside code. To control our pre-commit actions, we have made use of the hook library called *Husky*. Initially, we used Husky as a way to perform linting, i.e. the process of ensuring code consistency, and flagging code that contains bad practices. While the notion of a linter tool before each commit was nice in theory, we had much difficulty in implementing it correctly, eventually resulting in it being scrapped. The next step of our CI chain was to ensure that newly added code would not worsen the quality of our existing code. This was done using GitHub actions, which allowed us to run all tests that we had previously written (and which we describe in Section 11), and only allowing pushes to be merged if all tests pass. Our GitHub action script runs our unit and component tests in a cloud instance using Linux, while another cloud instance runs our end-to-end tests. Running our tests on an external set of operating systems was also done to ensure that our system would work independently of any operating system.

4.4 Use of Artificial Intelligence

In this project, we have made use of Artificial Intelligence (henceforth AI) tools to help with various problems throughout the project. AI has been a helpful companion throughout the project, where we have made use of the AI tools ChatGPT and Claude created by OpenAI and Anthropic, respectively (Anthropic, 2025; OpenAI, 2025). We primarily used AI at the beginning of the project, using it as a tool for understanding both TypeScript's and C#'s syntactical differences. We have strayed away from using direct translations between C# and TypeScript, since this would compromise with our goal of learning TypeScript through our spreadsheet implementation. Our use of AI tools has been limited to situations where documentation and/or examples of similar problems have been either hard to understand or difficult to find. In later stages of the project, we had a hard time keeping track of all the different interleavings that existed between class methods. Here we had luck with using AI to trace and recommend certain changes, which would catch tiny but consequential errors in our codebase. To exemplify this, we had a problem with getting support sets, i.e., the set of all cells that a cell supports, to work correctly. This was a problem we spent hours on debugging, and eventually, through the use of Claude were able to narrow the error down to the RefSet class (See Table 2). In this case, the use of AI acted similarly to that of pair programming, where we would supply the AI with a goal and context, and it would attempt to output a result. In some scenarios, however, we began to experience situations where AI tools began to hallucinate solutions that used non-existent code and libraries, or overall made the system perform worse than before. Our experience of AI has been twofold in this project. While in many cases, we have seen the helpful nature of working with AI tools in pair programming-like scenarios, but at the same time has spent much time reinvigorating prompts and providing additional context, which have yielded utterly useless results. In Section 12.2.2, we will reflect upon our AI usage.

4.5 Repository Setup

To conclude this section, we present an overview of our repository setup. Our repository has been stored in GitHub, which allows us to efficiently collaborate as a team as well as use version control through Git. GitHub allows for a multitude of different configurations to be made, such as branch protection rules, which have been an integral part of our repository management. To avoid any problems with the main branch, we decided to implement a branch protection rule, making direct pushes to the main branch disallowed, and instead requiring changes to be made through pull requests from separate branches, which had to be approved by at least one person. Our initial repository was created on ITU's own GitHub server. Later on, we decided to move onto the official GitHub server, since it allowed us to use more features, such as GitHub Actions. Furthermore, this move also made more sense, since our project was open-source after all. The addition of GitHub Actions and the introduction of continuous integration allowed us to ease our branch protection rules,

such that instead of someone having to manually approve a pull request, it could be done as long as the continuous integration chain passed.

5 Product Description

As a prelude to the forthcoming technical sections, we introduce our final implementation, named *TypeCalc version 0* (See Figure 7). This section provides an overview of both the graphical interface and the core functionality of our implementation. TypeCalc is a browser-based, open-source spreadsheet implementation built using the React library for its GUI and TypeScript for its backend. TypeCalc can be launched in all modern browsers using the npm package *Vite*, which is a popular web development tool for TypeScript and JavaScript applications (Vite, 2025), alongside the `npm run dev` command while being located in the project folder. Alternatively, TypeCalc can be accessed using the containerization tool *Docker* and *Docker-compose* using the guide listed in the repository READ-ME file (Docker, 2025). A link to the repository can be found here, on the front page of the report, or in the accompanying zip file.

When opening TypeCalc in the browser, the user is presented with a column and row 1-indexed setup. TypeCalc affords 65,536 columns and 1,048,576 rows, yielding more than 68 billion cells available for interaction in a singular sheet.

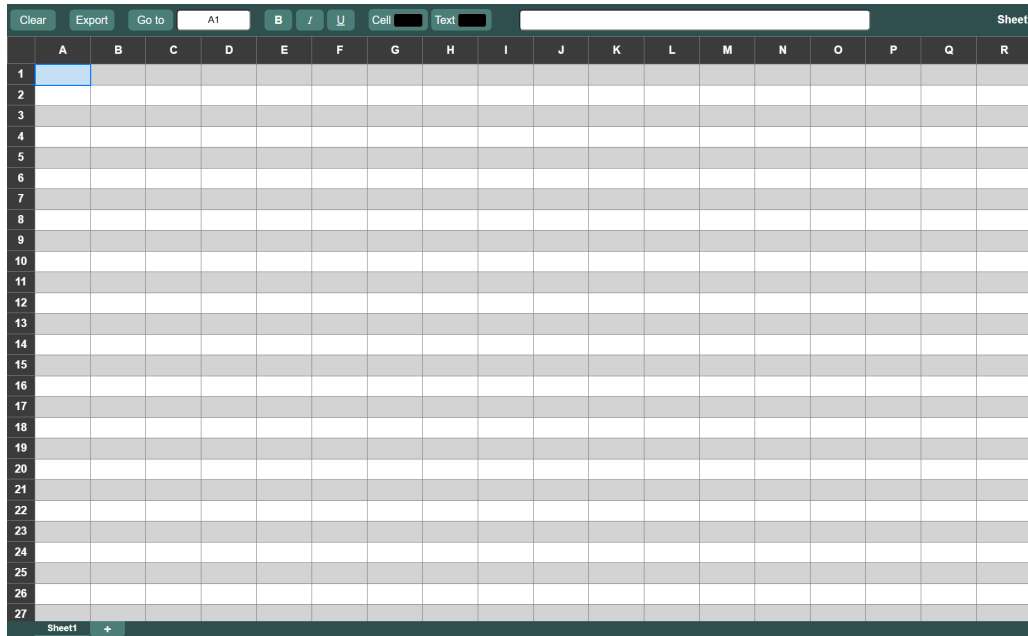


Figure 7: TypeCalc as it looks when opened up.

5.1 Cell Interaction

Each of the 68 billion cells can be interacted with, and accept one of four types of values: single-quotation strings, double-quotation strings, floating-point numbers, and formulas, which are characterised by the "=" at the beginning of their string (e.g. `=SUM(10,10)`).

A formula may contain constants, function calls, and arithmetic operators, as well as

	A	B	C	D
1				
2				
3				
4				
5			=A1 + B2 + C3	
6				

Figure 8: Example of cell references and their light-up feature, showcasing which cells they depend on. In the example chosen, the active cell depends on cell A1, B2, and C3, which are highlighted as a result of this.

references to other cells. Cell references in a formula can be written either in A1 notation, which is also used by our UI, or in R1C1 notation, which is commonly used in XMLSS, one of the custom XML file formats used in Excel. The implementation uses dynamic cell updates, meaning that if any changes are made to a referenced cell, it prompts a recalculation on all cells that depend on it. Cells that contain a formula with a cell reference will show the cell and/or cell area that they depend on through a green outline in the GUI. This helps users visualise the dependencies between cells, as seen in Figure 8. As is common for most spreadsheet implementations, and to allow users to work more efficiently, we have implemented key-bindings for undo (z), redo (y), copy (c), paste (v), cut (x), bold text (b), italic text (i), underlined text (u), and highlighting cell dependents (m) using the control-key + one of the aforementioned keys. The use of undo and redo follows the same semantic pattern as Excel, which is described in detail in Section 7.7. Copy and Cut are supported for multiple cells at a time, in which case, copying a formula that contains a reference will offset the cell reference in the new cell that it is pasted or cut to, such that =A1 becomes =B1 if copied to the cell to the right of the one containing =A1. Cell areas can be marked using the arrow keys while holding down the shift key. Cell areas are highlighted in a low opacity blue colour, showing the currently selected area (Figure 9). The **bold**, *italic*, and underlined formatting functions are used to style the text in the selected cell in the manner that their names imply. Each also has an associated button in the header that does the same, with the three being the only features in the header or footer that have a key-binding. We also added the ability for users to navigate between cells in the spreadsheet by using the arrow keys, as well as the Tab and Enter keys to move to the next column or next row, respectively.

	A	B	C	D	E
1					
2					
3					
4					
5					

Figure 9: Example of a marked cell area.

5.2 Sheet Selection

TypeCalc allows multiple sheets to work within a single workbook, as well as the possibility of creating new sheets within a workbook instance. To add a new sheet, the user can press the "+" key in the footer, which prompts the user to name a new sheet, which is then created and added to a separate tab in the footer (Figure 10). To switch between sheets, the user can press on the named sheets in this footer. Certain scenarios may require the user to work with cells that are located outside of their current sheet. In TypeCalc, this can be done using a sheet reference formula of the form "=sheetName!CellRef" as seen in Figure 11.

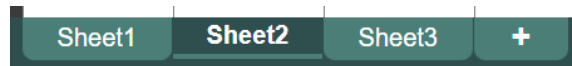


Figure 10: Sheet selection and creation menu.

	A	B	C
1			
2	10	20	30
3			
4			
5			
6			
7			

	A	B	C
1			
2			
3			
4			
5			
6			60
7			

Figure 11: Showcasing how cell references can exist between sheets. On the left, Sheet1 holds values in cells A2, B2, and C2, while on the right, Sheet2 sums the values in the cell C6.

5.3 Header Functionality

In the header, a number of features that are inspired by existing spreadsheet implementations have been made (Figure 12). For one, the header is a hub for information regarding the current cell. It provides the address of the currently active cell, as well as the formula of that cell in a formula box field that also allows the user to manipulate the cell directly. Additionally, the header provides UI elements for manipulating the cell text’s typographical emphasis, as well as changing both cell text colour and background colour. In this version of TypeCalc, we have not implemented a scroll bar, and to preserve a similar level of usability and to overall increase navigation in TypeCalc, we introduced the ”Go to” field, allowing users to move from one cell to any arbitrarily chosen cell in a fast manner. Finally, to reset the entire workbook, the user can press the ”Clear” button.



Figure 12: The sheet header.

5.4 Exporting and Importing files

In addition to offering its own functionality for working with spreadsheets, TypeCalc also works with the SpreadSheetXML (XMLSS) format (Microsoft, 2014). An XMLSS file can be directly dropped onto the TypeCalc GUI, which in turn imports the file and its values into the correct cells and calculates all the imported values. TypeCalc also offers the possibility of exporting a workbook with multiple sheets to the XMLSS format. This is done through a dropdown menu, which is accessed by hovering over the ”Export” button in the header. Combining this functionality allows users of TypeCalc to both save and import XMLSS files, allowing for an asynchronous workflow from a single user or potentially multiple users.

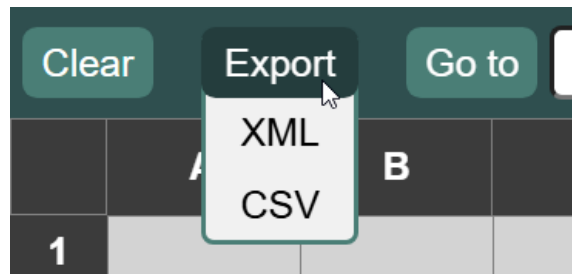


Figure 13: The dropdown menu that appears when hovering over the ”Export” button.

5.5 Benchmark

To test the efficiency of TypeCalc during larger recalculations, we have created a testing setup to benchmark the time it takes to update a large number of dependent cells. Our setup is as follows:

The cells in range B1:B10000 contain the numbers 1-10,000. This is accomplished via a 10,000-long dependency structure in which cell B2 contains the formula =B1+1, cell B3 contains the formula =B2+1 and so on. The cells in range C1:C10000 all contain a variation of the formula =SIN(\$A\$1 * B1) with B1 being a relative reference that is changed for every cell in the C column similar to the structure of the B column. The cell A1 is an editable cell for user input. The cell A2 contains the formula =SUM(C1:C10000). The file we used can be found in the code repository in the folder "extra_files" and is named "BenchmarkFile.xml". Be aware that loading this file into the application can take up to a minute.

The experiment works in the following way: Whenever a change is made to the A1 cell, all the cells in the C column have to call the SIN function on the new value of A1 multiplied by a corresponding value in a cell in the B column. This results in a total of 10,000 Formula.js (Section 8.1) SIN(...) function calls, 10,000 calls to our own MULTIPLY(...) function, 10,000 direct reads of the values in the B column (none of the cells here need to be recalculated and are always up to date) as well as a single call to the SUM(...) function, summing all 10,000 values of the C column.

We conducted this experiment using the =PI() formula and its resulting value in the A1 cell. In total, we made 10 tests, averaging 21.31 milliseconds for a standard minimal recalculation whenever A1 is changed. All of the test results can be seen in Figure 14.

D	E
	TIME
	22.5
	17
	17.9
	17.4
	30.1
	29.8
	25.2
	16.4
	20.2
	16.6
SUM:	213.1
AVG:	21.31

Figure 14: Benchmarking results. The unit of the TIME column is milliseconds.

Google’s performance model, RAIL (Google, 2025), specifies that users perceive responses in a browser as immediate if they take less than 100 ms. While our benchmark does not include other important processes like rendering, our virtualised sheet helps alleviate the time expenditure for this. On a qualitative level, these experiments seem to conclude instantaneously in the GUI.

6 Implementation

To start our technical deep-dive, we provide an overview of TypeCalc’s API-layer, how it manages our workbook, how it allows us to access back-end functionality such as instantiating cells through the front-end, and our workbook’s reading and writing of files. Lastly, we elaborate on our documentation for the open-source code repository.

6.1 Overview of Implementation

Our implementation of TypeCalc consists of a back-end and a front-end, which are connected via an API-layer. The back-end largely consists of code translations from CoreCalc with some modifications and added functionalities (Sestoft, 2014a). As the engine of TypeCalc, the back-end is responsible for creating a workbook, sheets and cells, and handling common spreadsheet functions. The front-end consists of code that pertains to the GUI of TypeCalc. It presents the outcome of the back-end in a local web browser that the user can interact with. This relationship is illustrated in Figure 15 where three folders are seen from the source code representing the front-end, API-layer, and back-end. In Sections 6.2, 7, 8, 9 and 10, we will elaborate on the characteristics and functionalities of the API-Layer, back-end and front-end.

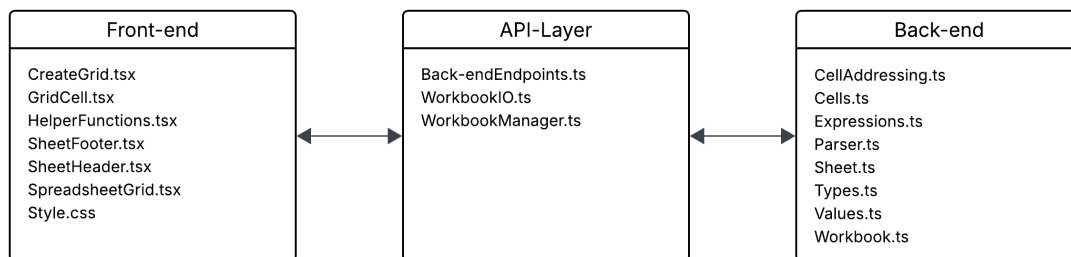


Figure 15: Visualisation of the three folders in the source code of TypeCalc. The three folders represent the front-end, API-Layer and back-end, with the files in their respective folder shown below the folder name. The arrows between the folders indicate the data flows between the front-end and API-layer, and between the API-Layer and back-end.

6.2 API-Layer

Before diving into the implementation details, we want to clarify our division between the front-end and the back-end of TypeCalc. We elected to create this split for a multitude of reasons. For one, it creates an easy and effective division of labour and tasks in the group, as we were able to decide on the endpoints between the two faces of our application, and then work from both ends in tandem. Secondly, it makes maintenance easier, not just for our project, but especially in anticipation of future work, where one could use our back-end for another project, such as a phone application or a software plug-in. In the following section, we will explain these integration points that we call the API layer. The

API layer can be seen in Figure 16, showcasing the three main parts of the API layer for TypeCalc.

For the purpose of this project, the front-end entails everything that has to do with the GUI. That is the rendering of cells, the display of values and formulas, front-end specific functions like text formatting, user input in the form of data entry, actions and navigation, and the ability to call certain back-end functionality, which includes the creation of new sheets and cells, recalculation and so on. The back-end, in turn, ensures that everything works with sheets and the workbook.

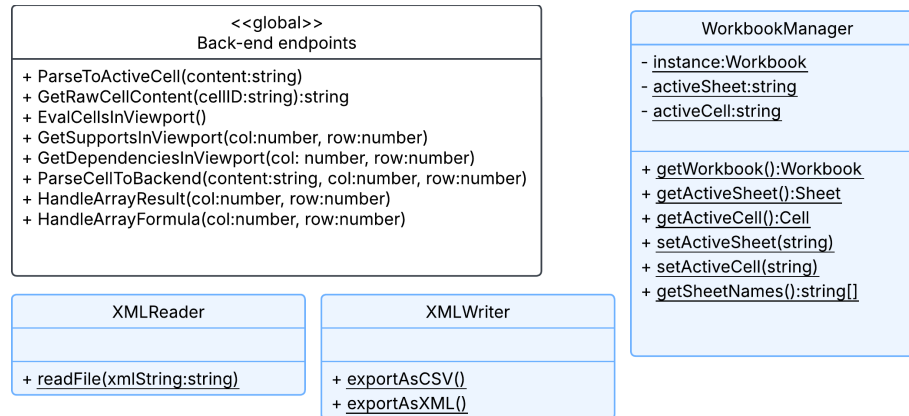


Figure 16: Overview of the API-layer classes WorkbookManager, XMLReader, XMLWriter, their fields, and their methods, as well as the global functions of the Back-endEndpoints.ts file.

6.2.1 Workbook Singleton Pattern

When dealing with the problem of interfacing between front-end and back-end, we weighed a number of considerations. Realising that our program does not need to support several workbooks in one instance of the application, we decided to create a static WorkbookManager class with a singleton workbook that has a lifespan of the entire running period of the program. This allows us to easily hook every sheet and cell change to this one workbook that we can reach from anywhere in the front-end. The WorkbookManager constantly keeps track of the active sheet and active cell, eliminating the need to pass around this information as method arguments. This also readily enables the functionality of writing in cells via the formula line, the text box at the top, similarly to how it works in Excel or other spreadsheets, and have it parse correctly to the last and currently active cell. The WorkbookManager class is a container for the workbook singleton, fields for the names and addresses of the currently active sheet and cell, and a host of methods for accessing and manipulating these fields, as seen in Figure 16. This singleton pattern may not be a great fit if the program were to run on a server and possibly support multiple workbooks at a time, in which case, one would have to create an alternative solution..

6.2.2 Cell Instantiation and Display

There are two sides to cell instantiation in our spreadsheet workbook. One is to write in a cell and have it be displayed in the GUI, and another is to have the cell represented in the back-end with all the necessary information. For the front-end specifically, one cell is always active. As mentioned, the active cell is the one that is lit up in a blue colour in the GUI, as seen in Figure 1, and is the one that is written in, whether you choose to write directly “in the cell” or instead in the formula text box at the top of the browser window. The information entered is then stored in two ways. Firstly, we store the exact string to be represented whenever a user refocuses this cell. This functionality mimics how Excel and other spreadsheets work – when you are outside of a cell, the cell shows its evaluated value (e.g. 20), but when you focus it again, it will display the original formula (e.g. =10+10). Going forward, the cell will only change its properties when this formula is changed. Separately from this, the formula string is then sent to the parser, which will instantiate a cell in the QT4 data structure and extract any information from the formula, such as the addresses of supporting cells, constants, functions and so on. The way we store the original formula string for formula cells is different from how this is handled in CoreCalc, which is explained in Section 12.1.2.

Whenever a cell is edited, the entire window in the front-end will query the back-end for the evaluated values of all cells in the current viewport using the function `EvalCellsInViewPort()`. Depending on screen size and zoom level, this is in the order of 1200 cells. In all tested scenarios, the necessary computations and subsequent rendering feel instantaneous, as previously described in detail. This could certainly be optimised, but we have witnessed no need to do so in the current iteration.

6.2.3 Workbook I/O

Another feature we will address in this section is file reading. For a spreadsheet program to be truly useful, it needs the ability to read files from other spreadsheet programs or from previous sessions. As briefly mentioned, we chose to use the XML-based format XMLSS that was introduced in the Microsoft Office package in 2003 (Microsoft, 2014). While not the most popular or modern spreadsheet format, this markup language solution allowed us to quickly set up a file reader that could use files from modern spreadsheets saved to this format. The structured language and pre-defined tags allow us to easily identify and parse data from these files into our own spreadsheet solution. To facilitate the transfer, we elected to make the entire grid area into a drag-and-drop zone using HTML. Whenever one of the XML files are hovered and dropped over the grid area, the `XMLReader` class in the `WorkbookIO.ts` file uses the third-party library `fast-xml-parser` to read through every line, parsing the sheet name, cell address and cell content into the cell format that we use in our back-end and insert them into the workbook singleton via the `WorkbookManager` (Gupta,

2025). These cells then simply constitute objects exactly as if they were created directly via the user interface and are rendered and displayed in the viewport as usual.

In CoreCalc, XML reading has an optimisation technique with which it recognises duplicate expressions and has every cell refer to a single instance of this expression instead of creating a new expression for every cell (Sestoft, 2014a). This is a great solution to optimise for memory, but since we elected to forego this optimisation in our cells in general, the XMLReader is also necessarily less advanced. This is an area with an obvious avenue for improvement, which we will explain in Section 12.1.2. It is worth noting that this optimisation renders no improvement in the file size when using XMLSS, as every cell and its formula are still represented separately in the XML file.

Having enabled file reading, we also determined that having a way to save a workbook was necessary and a natural extension of file reading, so that users could export their work to be imported and resumed later. The export feature consists of the two methods `exportAsCSV()` and `exportAsXML()`, which export to either of the two file formats. To enable them, we created a new method `iterateForExport()` in the class `SheetRep` (Table 2), which iterates through the QT4 data structure where our created cells are stored. While `SheetRep`'s iterator goes through the QT4 column by column, `iterateForExport()` instead goes row by row, i.e., first row 1, then 2, etc., rather than column A, then B, etc. This greatly simplifies the conversion to CSV or XML, as both file types store row-wise. `exportAsCSV()` keeps track of the current row and current column as well as which row and column the cell is in. We must keep track of the columns, as there must be a comma added between each cell in a row in the CSV format, while rows are tracked so that we can insert line breaks in the output whenever we find a cell in a new row. Representing the cell itself in the CSV output is its evaluated result, i.e., if cell B1 holds the expression `=A1` and A1 evaluates to 1, the CSV file would contain '1' rather than `'=A1'`, as seen in Figure 17. Therefore, the CSV format is not suitable for exporting files if the user wishes to maintain the formulas and cell references of the sheet.

	A	B	C
1	1		3
2	4	5	
3		8	9

workbook.csv
File Edit View
1, , 3
4, 5
, 8, 9

Figure 17: The sheet contents in TypeCalc (Left) and the same contents exported as a CSV file (Right).

The method `exportAsXML()` is a bit more complicated, as the XMLSS format requires the data to be formatted in a manner similar to the sections found in HTML (Figure 18). To accomplish this, we manually add any required sections before and after iterating through the workbook's sheets. Unlike the CSV format, the XMLSS format is able to take multiple sheets into account, so we must fetch every sheet in the workbook so that we can go through them one by one. For every sheet, we iterate through the QT4 and create a `<Cell>` element for each cell, specifying any required information. This includes its index (i.e., its column), its value, and its type, which determines whether its value should be read as a number or as text. Examples of such `<Cell>` elements can be seen in lines 24-26 and 29-31 of Figure 18. By default, the type is `String`, written to the output as `'String'`, since most of our various cell types' values should be read as strings. We perform a check in case the cell type is the single exception to this, i.e., if it is a `NumberCell` (see Table 2). If so, we instead write the cell's type as `'Number'` in the XMLSS output. During all of this, we escape special characters in the sheet's name and any `String` cells' value. We will elaborate further on `TypeCalc`'s different cell types in Section 7.2. We want to note that a late discovery showed that when exporting a workbook with multiple sheets, `exportAsXML()`'s exported file cannot be read by Excel, though it works for `TypeCalc`. We assume this is because Excel's requirements for reading an XMLSS file are stricter than our `XMLReader`'s.

```

1  <?mso-application progid="Excel.Sheet"?>
2  <Workbook xmlns="urn:schemas-microsoft-com:office:spreadsheet"
3      xmlns:o="urn:schemas-microsoft-com:office:office"
4      xmlns:x="urn:schemas-microsoft-com:office:excel"
5      xmlns:ss="urn:schemas-microsoft-com:office:spreadsheet"
6      xmlns:html="http://www.w3.org/TR/REC-html40">
7      <DocumentProperties xmlns="urn:schemas-microsoft-com:office:office">
8          <Author>TypeCalc Export</Author>
9          <Created>2025-05-30T12:55:00.729Z</Created>
10     </DocumentProperties>
11     <Styles>
12         <Style ss:ID="Default" ss:Name="Normal">
13             <Alignment ss:Vertical="Bottom"/>
14             <Borders/>
15             <Font/>
16             <Interior/>
17             <NumberFormat/>
18             <Protection/>
19         </Style>
20     </Styles>
21     <Worksheet ss:Name="Sheet1">
22         <Table ss:ExpandedColumnCount="1000" ss:ExpandedRowCount="1000" x:FullColumns="1"
23             ↪ x:FullRows="1">
24             <Row ss:Index="1" ss:AutoFitHeight="0">
25                 <Cell ss:Index="1">
26                     <Data ss:Type="Number">123</Data>
27                 </Cell>
28             </Row>
29             <Row ss:Index="2" ss:AutoFitHeight="0">
30                 <Cell ss:Index="2">
31                     <Data ss:Type="String">=A1</Data>
32                 </Cell>
33             </Row>
34         </Table>
35     </Worksheet>
36 </Workbook>

```

Figure 18: Example of XMLSS output with two cells. A1 is a NumberCell evaluating to 123 and B2 holds the cell reference =A1.

6.3 Documentation

As previously stated, one of our learning outcomes includes documenting the implemented functionality and our design choices for the open-source project. To document the implemented functionality, we have inserted *JSDoc* directly into the source code. JSDoc is a code documentation generator for JavaScript, used by developers to describe the functionality of their source code, and it is also supported in TypeScript (Microsoft, 2025c; Williams et al., 2025). JSDoc works by inserting text between the start indicator `/**` and end indicator `*/`, and a good practice is to insert the comments immediately above a piece of code (typically a class, method, or test) (Williams et al., 2025). Different JSDoc tags can be used, such as `@constructor` to indicate that the following code is a constructor, or `@param` to highlight the name and meaning of a parameter. An example of JSDoc from the source code of TypeCalc is shown in Figure 19. We made use of the TypeDoc package to wrap all of the methods alongside their documentation available as HTML files, making it easier to get an overview of the classes, methods, and functions as a whole. This HTML file can be found within the GitHub repository within the docs folder. Regarding documentation

of our design choices, we have not inserted this into our source code. Instead, we use this report as documentation of our design choices because it requires more thorough reasoning and descriptions than what JSDoc is suited for. Therefore, this report will be attached as a PDF to the final open-source repository for future interested users and contributors.

```
/**
 * Evaluates the cell's expression and caches its value.
 * Detects Cycles and reports these as errors.
 * It uses the current state of the Cell to determine the need for evaluation.
 *
 * @param {Sheet} sheet - The sheet where the cell exists
 * @param {number} col - The column of the cell
 * @param {number} row - The row of the cell
 * @returns {Value} The result of the formula evaluation. Returns
 * ↪ ErrorValue.cycleError if a cyclic dependency is detected.
 */
public override Eval(sheet: Sheet, col: number, row: number): Value {
    // Method implementation...
}
```

Figure 19: JSDoc for the `Formula.Eval()` method. The first three lines describe the purpose of the method, while the following lines use the JSDoc tags `@param` and `@returns` to describe the parameters and return type of the method.

7 Back-end

Following our explanation of the API-layer, we will here elaborate on the key parts of our back-end. These parts include cells, values, expressions, sheets and workbook, cell addressing and referencing, our undo/redo feature's functionality, and recalculation. In doing so, we also provide an overview of our various back-end classes and a description of each of them.

7.1 Overview of Back-end

The back-end is the calculation engine of TypeCalc. It is where the data structure is built and maintained, the cells of a sheet are stored, and all formulas are calculated. In this section, we will elaborate on the most important aspects of the back-end, but before that, we want to provide an overview of the relevant classes with the UML class diagram seen in Figure 20. Along with the diagram, Table 2 describes the purpose of each class. All classes are shown in the diagram, except three classes called RefSet, Adjusted, and SpreadsheetLexer. That is because they were only part of dependency relations, which we decided not to visualise to avoid clutter, but they are described in Table 2. TypeCalc's UML class diagram looks different from CoreCalc's because we have added some classes, and because some relations from CoreCalc were not visualised in the class diagram from SIT (Sestoft, 2014b). However, we have followed the same UML rules as SIT, for which we list the definitions here:

- **Roman font:** Represents a concrete class.
- **Italic font:** Represents an abstract class.
- **Asterisk (*):** Means zero to many.
- **Triangular arrow:** Represents inheritance.
- **Open rhombus:** Represents aggregation where the object at the open rhombus end references the object at the other end.
- **Solid rhombus:** Represents a stronger form of aggregation called composition, where the object at the solid rhombus end "owns" the referred-to object.

The UML class diagram consists of a hierarchy with different groups of classes. The groups correspond with the file names of the back-end code and are listed below:

- **Workbook.ts:** Workbook.
- **Sheet.ts:** Sheet and SheetRep.
- **Cells.ts:** *Cell*, *ConstCell*, BlankCell, NumberCell, QuoteCell, TextCell, BooleanCell, Formula, ArrayFormula, and CachedArrayFormula.
- **Expressions.ts:** *Expr*, *Const*, NumberConst, TextConst, ValueConst, BooleanConst, Error, ExprArray, FunCall, CellRef, and CellArea.
- **Types.ts:** CyclicException, ImpossibleException, NotImplementedException, Formats, and ValueCache.

- **Values.ts:** *Value*, *ErrorValue*, *TextValue*, *NumberValue*, *BooleanValue*, *ArrayValue*, *ArrayView*, and *ArrayExplicit*
- **CellAddressing.ts:** *Interval*, *SuperRAREf*, *A1RAREf*, *R1C1RAREf*, *SuperCellAddress*, *RAREfCellAddress*, *A1RefCellAddress*, *FullCellAddress*, *SupportSet*, *SupportCell*, and *SupportArea*.
- **Parser.ts:** *SpreadsheetParser* and *SpreadsheetVisitor*.

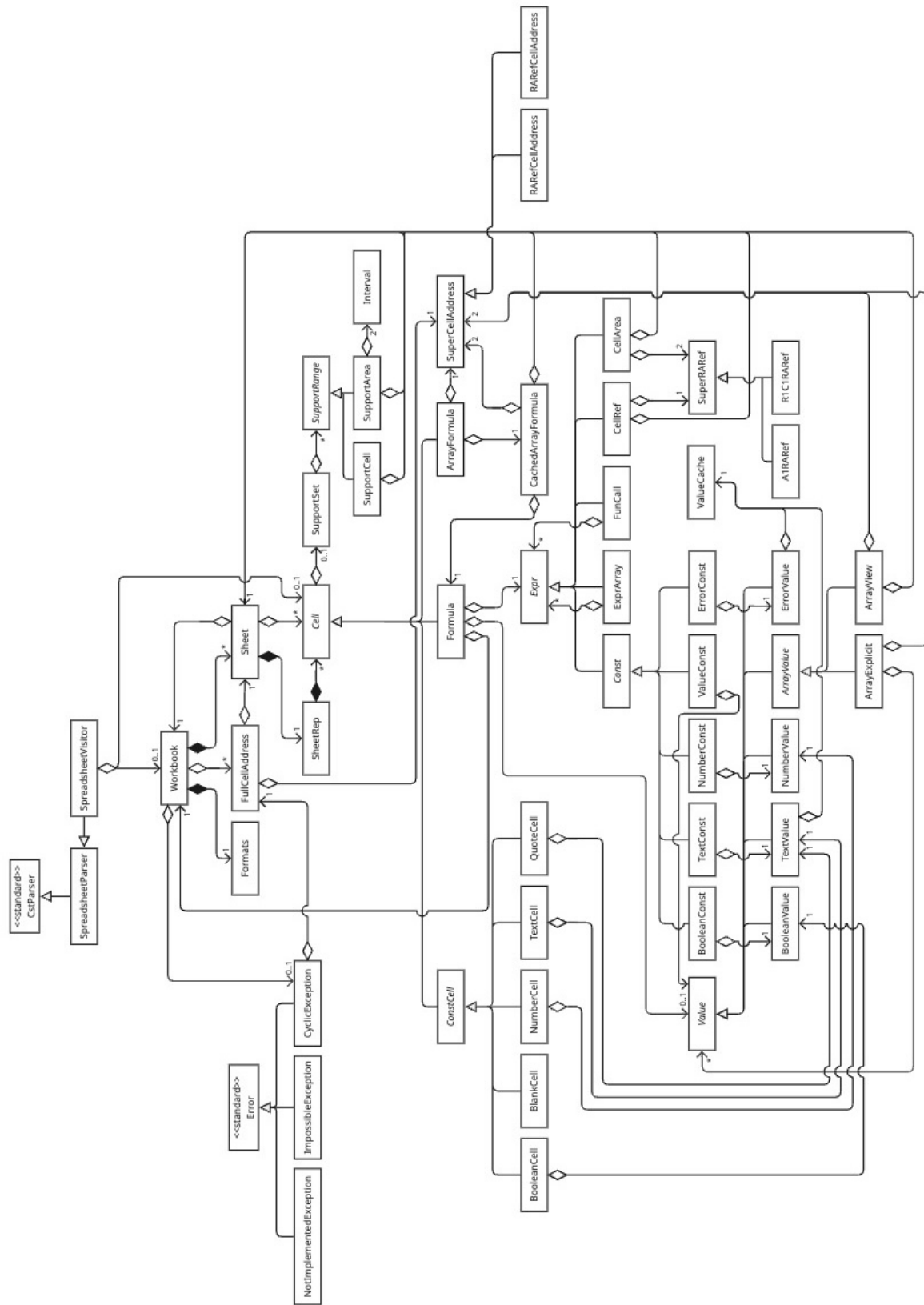


Figure 20: UML class diagram visualising the back-end of TypeCalc.

Class Name	Description
A1RAREf	Handles cells of the A1 cell reference notation. Preserves relative and absolute rows and columns. Parses the cell reference from string to the A1 format.
A1RefCellAddress	The cell address for an A1 format.
Adjusted	A class used for insertion of new rows and columns. In this version of TypeCalc it is not being used.
ArrayExplicit	A type of ArrayValue. It caches values of a two-dimensional array, which are the results of a formula that returns an array-valued result.
ArrayFormula	A type of Cell that references a CachedArrayFormula to retrieve one of its values.
<i>ArrayValue</i>	A type of Value that represents a two-dimensional array of values in a sheet. Defines functionalities such as getting and setting cells in an array, and viewing and comparing array contents.
ArrayView	A type of ArrayValue. It references a two-dimensional array in a sheet to view the cells within this area.
BlankCell	A type of ConstCell that is used for representing an empty cell in a support set.
BooleanCell	A type of ConstCell that holds a BooleanValue.
BooleanConst	A type of Const that holds a BooleanValue in a formula.
BooleanValue	A type of Value that holds an instance of the primitive <code>boolean</code> type.
CachedArrayFormula	Caches a formula that outputs an array result and references the location in the sheet where the result should be.
<i>Cell</i>	Represents the different cell types of TypeCalc. It defines a structure for relevant cell functionalities such as evaluation, dependencies, and support sets.
CellArea	A type of Expr that references a two-dimensional area of a sheet in a formula. When evaluated, it returns an ArrayView representing this area.
CellRef	A type of Expr that references a single cell in a formula.
<i>Const</i>	A type of Expr that represents an immutable expression in a formula.
<i>ConstCell</i>	A type of Cell that represents an immutable cell in the sheet.
CstParser	A class responsible for creating a concrete syntax tree in Chevrotain.
CyclicException	A type of the native JavaScript Error class. It is thrown if a cyclic dependency occurs, and it holds a reference to the culprit cell. This has been deprecated as described in Section 7.8.
Error	The native Error class creates error objects that are thrown when runtime errors occur (Mozilla Developer Network, 2025b).
ErrorConst	A type of Const that is used when a formula contains an error. The type of the error can vary and is represented via an ErrorValue reference.
ErrorValue	A type of Value that represents an error condition in a cell, such as a reference error, where a formula references a non-existent cell.
<i>Expr</i>	Expr and its subclasses are used to build expressions recursively. Expressions are the building blocks of formulas, such as function calls, constants, and cell references.
ExprArray	A type of Expr that holds an array of multiple expressions.
Formats	Determines the formatting settings of the Workbook. This includes the reference style (e.g. A1).
FullCellAddress	Provides a complete cell address that includes both the location of a cell with a SuperCellAddress reference and which sheet the cell is in with a Sheet reference.
FunCall	A type of Expr that calls the functions of TypeCalc, such as SUM, ADD, and IF.

Continued on next page

<i>Continued from previous page</i>	
Class Name	Description
ImpossibleException	A type of the native JavaScript Error class. It is thrown if the internal logic of TypeCalc is violated. Ideally, it should never be thrown.
Interval	An interval represents a contiguous range of rows or columns defined by a minimum and a maximum bound.
NotImplementedException	A type of the native JavaScript Error class. It is thrown if a method is defined but not implemented yet.
NumberCell	A type of ConstCell that holds a NumberValue.
NumberConst	A type of Const that holds a NumberValue in a formula.
NumberValue	A type of Value that holds an instance of the primitive number type.
QuoteCell	A type of ConstCell that holds a TextValue. QuoteCell represents a single-quoted string constant.
R1C1RAREf	A reference to a cell using the R1C1 format.
RAREfCellAddress	A type of SuperCellAddress for handling relative or absolute cell references.
RefSet	A set of CellRefs and CellAreas already seen by a VisitRefs visitor to avoid adding duplicate cell references to support sets.
Sheet	A rectangular collection of cells that belongs to a Workbook. A sheet holds an instance of SheetRep where the cells are cached.
SheetRep	SheetRep provides an abstraction for the QT4 quadtree.
SpreadsheetLexer	SpreadsheetLexer defines lexical tokens using regular expressions.
SpreadsheetParser	SpreadsheetParser defines rules for processing lexical tokens and generates a syntax tree.
SpreadsheetVisitor	SpreadsheetVisitor traverses a syntax tree and provides semantic actions to the input.
SuperCellAddress	Stores the coordinates of a spreadsheet cell using row and column numbers (C0R0 notation).
SuperRAREf	Represents the relative/absolute status of a cell reference.
SupportArea	A type of SupportRange that represents a two-dimensional area of a sheet that another cell depends on.
SupportCell	A type of SupportRange that represents a single cell in a sheet that another cell depends on.
<i>SupportRange</i>	Represents a range of cells that support another cell.
SupportSet	Stores a list of cells that depend on a given cell.
TextCell	A cell that maintains a TextValue.
TextConst	An expression, which belongs to a TextCell.
TextValue	A value consisting of a string.
<i>Value</i>	An abstract class used to describe value types.
ValueCache	A cache that ensures only one object exists for each unique key, avoiding duplicate instances.
ValueConst	A ValueConst is an arbitrary constant-valued expression.
Workbook	A representation of multiple sheets, and the component responsible for recalculation of cells

Table 2: Descriptions of the classes in TypeCalc's back-end.

7.2 Cells and SupportSets

Cells form the foundation of the TypeCalc implementation, where they serve as a discrete computational unit that can be added to a sheet and later appear in the GUI. Our implementations of the Cell classes are based on CoreCalc cell architecture and provide specific cell types for different types of values. Cells are generated through the Chevrotain parser, which we describe in Section 9. To structure the cell architecture, we utilise an abstract superclass Cell (Table 2), which provides a variety of concrete and abstract methods, ensuring consistency between subclasses of Cell. Within TypeCalc, we employ three primary cell types, each of them designed for specific situations, and each inheriting from the Cell superclass. A common denominator for all cells is that they possess a column and a row, a supportSet, and a textField to indicate their formula text, the latter of these being mostly relevant for the front-end. The supportSet of a cell is the list of cells that depend on that cell. The supportSet is of type SupportSet, which holds a ranges array of type SupportRange. The support range can include either a SupportCell or a SupportArea, and is explained in Table 2. Support sets are an essential part in ensuring that each cell that depends on another cell maintains an updated value at all times. We touch more upon this concept in Section 7.8.

The first cell type we describe is ConstCell, which is used for static data retrieval. They are the most basic version of a cell in TypeCalc, maintaining only their value field and no expression. If probed for evaluation, they will simply return their value field. There are five specialised classes that inherit from the ConstCell superclass: NumberCell, BooleanCell, TextCell, QuoteCell, and BlankCell. All of these specialised cells maintain an inner value related to their namesake, except for BlankCell, which, when prompted, instead returns a null pointer. The reasoning for the existence of a BlankCell is to maintain a consistent supportSet when another cell refers to a blank cell. Examples of ConstCell values include 10, true, and "hello world".

The second cell type is Formula cells (Table 2). A Formula cell contains two additional fields compared to a ConstCell: an expression *e* and a CellState *state*. A Formula cell can be in one of four states at a time, which denote the current need to evaluate the expression within the cell. Expressions will be described in Section 7.4. The four states of the CellState enum are: *Dirty*, *Enqueued*, *Computing*, and *Uptodate* (Sestoft, 2014b). If a cell is Dirty or Enqueued, it needs to be evaluated, while a cell in a computing state is currently being evaluated. Finally, once the cell has finalised its computation, it is marked as Uptodate. The need for a Computing state is to ensure that cyclic dependencies are handled during recalculation. That is, if a formula cell tries to evaluate its expression while being in the Computing state, and the expression involves a cell reference that leads back at the same cell, then a cyclic dependency will be discovered because we cannot evaluate computing cells (see line 5-10 in Figure 21) (example from GUI in Figure 2). We will address the aspect of recalculation in-depth in Section 7.8. On initialisation, Formula cells are created in the Uptodate state, but when they are added to a sheet, they will be marked as Dirty (Sestoft, 2014b). Upon evaluation, they will enter the switch case seen in line 2 of Figure 21. If the cell is marked in the Dirty state or the Enqueued state, a subsequent reevaluation of its expression is performed, setting state to Uptodate, and probing cells in their supportSet to recalculate their expressions as well. A Formula cell type is characterised by expressions that have an equals prefix (=) in the user input, followed by an expression. Examples of formulas are: =10, =10*B5, ="hej", and =SUM(10, 10).

```

1 public override Eval(sheet: Sheet, col: number, row: number): Value {
2     switch (this.state) {
3         case CellState.Uptodate:
4             break;
5         case CellState.Computing:
6             const culprit: FullCellAddress = new FullCellAddress(sheet, null,
7                 ↪ col, row);
8             const msg = `### CYCLE in cell ${culprit} formula ${this.Show(col,
9                 ↪ row, this.workbook.format)} `;
10            const err = ErrorValue.Make("#CYCLE!");
11            this.v = err;
12            return this.v;
13        case CellState.Dirty:
14        case CellState.Enqueueed:
15            this.state = CellState.Computing;
16            this.v = this.e.Eval(sheet, col, row);
17            this.state = CellState.Uptodate;
18            if (this.workbook.UseSupportSets) {
19                this.ForEachSupported(
20                    Formula.EnqueueCellForEvaluation);
21            }
22            break;
23    }
24    return this.v as Value;
25 }

```

Figure 21: The `Formula.Eval()` method, and how cell states are used in the evaluation of a formula cell type. In addition to this, it should be noticed how a change in the formula value will set off supporting cells' evaluation step.

The third cell type is `ArrayFormula` cells, which are used when a function call returns an array-valued result. Here, `ArrayFormula` cells are used to denote a singular element of the array-valued result. To avoid excessive recalculations for every `ArrayFormula` cell, we rely on a `CachedArrayFormula`. `CachedArrayFormula` is an implementation from `CoreCalc` that evaluates the expression of a `Formula` cell once, and then informs the `ArrayFormula` cells which value they should hold each (Sestoft, 2014b). `ArrayFormula` cells are categorised in the same way as `ConstCells`, but internally maintain a reference to the `CachedArrayFormula`, ensuring they are updated as needed.

7.3 Values

Each cell contains a value field, which denotes either an evaluated value from a formula or alternatively the constant value of a `ConstCell` (Table 2). In `TypeCalc`, values are maintained in a number of specialised classes: `NumberValue`, `ErrorValue`, `TextValue`, `ArrayValue`, and `BooleanValue`, which all inherit from an abstract superclass `Value`. These specialised classes act as abstractions from their namesake primitive types. They are mentioned in Table 2.

While the implementation of the `Value` classes is based on `CoreCalc`, there are some significant changes to `TypeCalc` due to the type system differences between `TypeScript/JavaScript` and traditional languages. Traditional object-oriented languages, such as `Java` and `C#`, provide multiple numeric types using 8, 16, 32, 64-bit representations with further divisions on integers, and floating point values, unsigned and signed (Bill, 2022). Similarly, a number of `String` types exist for `C#` (Bill, 2024). In `TypeScript` we have less specialised types, utilising only number type to represent

all aforementioned numeric values using 64-bit representations, and `String` type representations of strings and characters. While we will discuss the consequence of these different type systems in Section 12.1.1, the immediate result of rewriting the value subclasses in TypeScript meant that most of the conversion methods from CoreCalc could be merged together. This resulted in us creating the `FromNumber()` method to contain all variants of a number, and `FromString()` for all strings in TypeCalc.

7.4 Expressions

As noted in Section 7.2, formulas maintain expressions that can be evaluated. In TypeCalc, expressions are organised into hierarchical class structures, which we describe using three categorisations. These expression class structures are subclasses of an abstract class `Expr`, which defines the overarching functionality for each expression class. In particular, the `Expr.Eval()` method is a highly relevant part of all expression classes, as it returns the actual value that will be displayed in the GUI, as well as setting any `FunCall` expressions to be reevaluated.

7.4.1 Constant expressions

The first category of expressions is constant expressions; expressions that maintain a constant value after initialisation. This category includes the superclass `Const`, which inherits from `Expr`, and five specialised classes: `NumberConst`, `TextConst`, `ValueConst`, `ErrorConst`, and `BooleanConst`, all inheriting from the `Const` class. Each constant expression will return its value field when prompted for evaluation. The reasoning for doing this is that other cells may require the result of a const expression for their expressions to be evaluated. While our implementation of expressions is based on the CoreCalc implementation, we added the expression class `BooleanConst` ourselves. We did this because during parsing, we encountered a problem with registering a boolean statement (`true`, `false`), which the parser defaulted to a `TextConst`. Thus, by resolving this problem with `BooleanConst`, TypeCalc supports the use of boolean values in the spreadsheet.

7.4.2 Reference expressions

The second category comprises reference expressions. This category includes the specialised classes `CellRef` and `CellArea` used for single-cell references and multi-cell references, respectively. Reference expressions point towards other cells and use their evaluated values for their own inner values. Reference expressions are responsible for populating the support set of a cell, based on the contents of their expression. To exemplify this, please consider the example in Figure 22, which shows the cells that depend on the cell located in A1.

	A	B	C	D	E	F	G	H	I	J
1										
2										
3				=A\$1	=A\$1	=A\$1	=A\$1	=A\$1	=A\$1	
4				=A\$1	=A\$1	=A\$1	=A\$1	=A\$1	=A\$1	
5				=A\$1	=A\$1	=A\$1	=A\$1	=A\$1	=A\$1	
6				=A\$1	=A\$1	=A\$1	=A\$1	=A\$1	=A\$1	
7										
8										
9										

Figure 22: Using CellRefs in TypeCalc. The cells with a blue outline depend on the cell located in A1. In other words, A1’s supportset contains all of the outlined cells.

While the CellRef class is in charge of a single cell reference, CellArea is used to reference an area consisting of cells, by referencing two cell references, such as =A1:A3. In the case where we reference a non-existing cell, we create a BlankCell on the referenced cell location, and add the referencing cell to its supportSet field. This helps maintain the correct support set, in case the non-existing cell is updated via user input.

CoreCalc has a valuable memory optimisation in how it allows multiple formulas to use a single reference expression. In TypeCalc, we ended up omitting this feature. We will describe the differences between the two implementations and our thoughts on this topic in Section 12.1.2.

7.4.3 Function expressions

The final category of expressions is function expressions, which contain the two classes: FunCall and ExprArray. While we provide a detailed description of these classes in Section 8, we will shortly describe FunCall and ExprArray here. The FunCall class is used to handle function calls, consisting of a function and zero or more expressions. FunCalls can contain nested expressions, meaning that a valid FunCall expression can be: =SUM(SUM(10,10),SUM(20,20)). The aforementioned example will then consist of a function identifier (e.g. SUM), containing two nested FunCall expressions. The introduction of the ExprArray class was done in an attempt to capture expressions that consist of array values, such as the FREQUENCY function, which requires =FREQUENCY(A1:A9,[2;4]) to work with Formula.js

7.5 Workbook and Sheets

With a better understanding of the interplay between cells, expressions, and values, we describe the broader context in which all these classes are used together in a sheet. The Sheet class provides a grid-like abstraction for working with the QT4 data structure in TypeCalc (described in Section 7.5.1). It is responsible for the manipulation and access to cells within the QT4 data structure. This includes the ability to insert, remove, move, paste, undo, redo and retrieve cells:

- `SetCell()` and `Get()`: `SetCell()` inserts a new cell into the sheet, overwriting existing cells in the location, and ensuring their support set is transferred to the new cell. The `Get()` method retrieves a non-null cell from a column and row index if a cell exists there.

- `ForEachInArea()`: Performs an operation on all non-null cells located within a self-defined area in a sheet.
- `CutCell()` and `PasteCell()`: `CutCell()` is used to move a non-null cell from a sheet to another cell location within a sheet, whereas `PasteCell()` adds a new cell to the sheet. These methods implement the same functionality as their `CoreCalc` equivalent, instead relying on regex for manipulating formula reference strings.
- `Undo()` and `Redo()`: By maintaining a history of cell changes, `Undo()` and `Redo()` allows revisions on the sheet to be done. We describe this in depth in Section 7.7

The `Workbook` class maintains one or more sheets and is used to register cell changes in its sheets, and handle sheet recalculation efforts, which we describe in Section 7.8. Compared to a sheet, the `Workbook` class provides information regarding volatile cells, sheets, and potential cyclic dependencies.

- `Recalculate()`: Sets all changed cells to the dirty state, and subsequently evaluates them.
- `IncreaseVolatileSet()` and `DecreaseVolatileSet()`: keep track of volatile cells, which need to be evaluated every time a cell is changed.

7.5.1 QT4 Data structure

A consideration of paramount importance in relation to spreadsheet implementation is the choice of data structure to represent the possibly millions of cells in a sheet. An emphasis on this aspect of TypeScript is part of our intended learning outcomes and is also an area in which we find many sub-optimal solutions in existing open-source implementations. For example, `react-spreadsheet` uses a simple 2D-array to store all its cells (Aaronsohn, 2025), while others elect to use different hash tables to store data (Handsontable, 2025a). These implementations are, interestingly, incredibly similar due to how the default JavaScript/WebAssembly engine works in popular browsers. Chrome's V8 Engine, for example, will optimise arrays based on their level of density (Bynens, 2017). For dense arrays, it uses the contiguous memory array style that we are accustomed to when working with programming languages. If the array becomes too sparse, however, it will change to a dictionary-style table with the indices as strings for keys. This means that memory usage is generally well-optimised in JavaScript apps, however, this division is still not necessarily optimal for spreadsheets. One can imagine a normal spreadsheet that has a very "clumpy" structure - some areas are very densely populated with data, while others are essentially completely empty. Such a spreadsheet will have suboptimal efficiency with either of the aforementioned structures.

Our solution instead makes use of the QT4 simplified quadtree. This data structure, invented by Peter Sestoft (Sestoft, 2014b), hosts a 4-layer repeating subdivision of the entire spreadsheet, capable of representing 65536 columns and 1048576 rows. On the abstract level, the top layer contains 512 tiles (16 columns x 32 rows), all of which themselves contain 512 tiles, all the way down to the bottom layer, which hosts a reference to every single cell. Of course, if this were translated directly, we would simply have an array with an overhead, but what makes this structure very useful for our application is the fact that reality is different from abstraction. Instead, at the creation of the initial, empty sheet, only the top layer, an array of length 512, exists. Trying to query for any cell will simply return a null pointer, as nothing else exists in the array or elsewhere. Once a

cell is declared and added to the sheet, the top layer will instantiate the necessary tile - if the cell has the address B2, for example, the top tile will create one new 512-sized tile and reference it at the first index of the top layer tile. The same thing happens in this new tile, and the subdivision repeats until we reach the bottom (fourth) layer, in which the actual cell reference is placed. To place one cell, we thus need a total of 4 arrays of length 512, as well as the singular cell reference. If another cell is added right next to it, we need no further allocation - this cell is simply placed in the existing tile. If, however, the cell is placed in another area, we will need between one and three new tiles, depending on how far away this new cell is placed. If the sheet is fully populated, this will work similarly to a two-dimensional array, but with a small overhead. However, it is extremely uncommon for a spreadsheet to contain the full 68 billion cells that it can theoretically host, and even more unusual that anyone would attempt to modify such a dataset in a spreadsheet program. Instead, the data structure is very efficient for clumpy spreadsheets where data is located in clusters around the larger cell area.

Another notable strength of this data structure is how we can index and find cells inside it. Since every subdivision's column and row counts are powers of two, we can calculate indices by bitwise operations and shifts. In Figure 23 we see the `Get()` method for retrieving a cell from the QT4 representation. The variables that are used in the method are:

- `c` and `r`, which are the column and row of the cell we want to retrieve.
- `LOGW` and `LOGH`, which are constants used to determine the tile size.
- `SIZEW`, which is the maximum number of columns in the sheet.
- `SIZEH`, which is the maximum number of rows in the sheet.
- `W`, which is the number of horizontal partitions (or width) of a single tile in the QT4.
- `H`, which is the number of vertical partitions (or height) of a single tile in the QT4.
- `MW`, which is the mask `0...01111` with `LOGW` 1-bits, used for bitwise `&` to avoid the use of modulo when finding the tilewise column index.
- `MH`, which is the mask `0...011111` with `LOGH` 1-bits, used for bitwise `&` to avoid the use of modulo when finding the tilewise row index.

For example, if we wanted to retrieve the cell Z98 from the QT4 sheet representation, we would use the `Get()` method seen in Figure 23, which is derived from Sestoft's implementation in *CoreCalc* (Sestoft, 2014b). We continue this example below the figure.

```

1      public Get(c: number, r: number): Cell | null {
2          if (c < 0 || this.SIZEW <= c || r < 0 || this.SIZEH <= r) {
3              return null;
4          }
5          const tile1: Cell[][][] | null[][][] =
6          this.tile0[(((c >> (3 * this.LOGW)) & this.MW) << this.LOGH) |
7          ((r >> (3 * this.LOGH)) & this.MH)];
8          if (tile1 == null) {
9              return null;
10         }
11         const tile2: Cell[][] | null[][] =
12         tile1[(((c >> (2 * this.LOGW)) & this.MW) << this.LOGH) |
13         ((r >> (2 * this.LOGH)) & this.MH)
14         ];
15         if (tile2 == null) {
16             return null;
17         }
18         const tile3: Cell[] | null[] = tile2[(((c >> this.LOGW) & this.MW)
19         << this.LOGH) |
20         ((r >> this.LOGH) & this.MH)
21         ];
22         if (tile3 == null) {
23             return null;
24         }
25         return tile3[(c & this.MW) << this.LOGH | (r & this.MH)];
26     }

```

Figure 23: The code for getting a cell in the QT4 simplified quadtree using bit operations

First, we determine the row and column of this cell. Suffice to say that this is always done as part of addressing the cell, and we can easily find that the cell belongs to row 97 and column 25 based on an index that starts at 0. Next, we determine the values of the different fields to ease the following operations.

$$\begin{aligned}
 LOGW &= 4 \\
 W &= 1 \ll 4 = 16 \\
 MW &= 15_{10} = 1111_2 \\
 SIZEW &= 1 \ll (4 * 4) = 65536 \\
 LOGH &= 5 \\
 H &= 1 \ll 5 = 32 \\
 MH &= 31_{10} = 11111_2 \\
 SIZEH &= 1 \ll (4 * 5) = 1048576 \\
 c &= 25_{10} = 11001_2 \\
 r &= 97_{10} = 1100001_2
 \end{aligned}$$

Now, following line 2 in Figure 23 we check whether the indices we are looking for are within the bounds of the sheet representation. Since $25 < 65536$ and $97 < 1048576$, this cell is within the allowed boundaries. We move on to determine which first-layer tile contains the cell.

$$\begin{aligned}
 tile1 &= tile0[(((11001_2 \gg 12_{10}) \& 1111_2) \ll 5_{10}) | ((1100001_2 \gg 15_{10}) \& 11111_2)] \rightarrow \\
 tile1 &= tile0[0 | 0] = tile0[0]
 \end{aligned}$$

The cell Z98 is thus placed in the tile at index 0 of the outermost tile in the QT4 structure. We

repeat similar operations to further pinpoint the cells' placement.

$$tile2 = tile1[(((11001_2 >> 8_{10}) \& 1111_2) << 5_{10}) \mid ((1100001_2 >> 10_{10}) \& 11111_2)] \rightarrow$$

$$tile2 = tile1[0 \mid 0] = tile1[0]$$

The cell is thus placed in the tile at index 0 of the second tile we just found.

$$tile3 = tile2[(((11001_2 >> 4_{10}) \& 1111_2) << 5_{10}) \mid ((1100001_2 >> 5_{10}) \& 11111_2)] \rightarrow$$

$$tile3 = tile2[100000 \mid 11] = tile2[100011_2] = tile2[35_{10}]$$

The cell is thus placed in the tile at index 35 of the third tile we found previously. Lastly, we find the cell's index in this last tile, which is simply an array of 512 cell references.

$$cellref = tile3[(((11001_2 \& 1111_2) << 5_{10}) \mid (1100001_2 \& 11111_2))] \rightarrow$$

$$cellref = tile3[100100000_2 \mid 1_2] = tile3[100100001_2] = tile3[289_{10}]$$

Or:

$$cellref = tile0[0][0][35][289]$$

And we can then return this cellref. Any other cell in the QT4 can be found in a similar fashion through bit shifts and operations, completely removing the need to do division and modulo operations, which would otherwise be the intuitive way of partitioning the subspaces.

7.6 Cell Addressing and Referencing notations

A crucial part of the TypeCalc implementation relates to cell locations. Each cell, which has been inserted into a sheet, corresponds to a cell address, and can then be referenced using a cell reference. In many popular spreadsheet implementations, cell addresses are based on a letter-and-number system, signifying the column as one or more letters and the rows as a number; known as A1 notation. For instance, the cell address B7 will be registered as the second column and the seventh row. Additionally, the use of the R1C1 notation is also widely used, which first denotes the row and then the column of a cell address. The cell at location B7 in R1C1 notation would be referenced R7C2. Within TypeCalc, we allow the use of A1 notation using the A1RefCellAddress class and R1C1 notation using the R1C1RefCellAddress class. The decision to split these into more specialised classes is reflected upon in Section 12.1.1. Cell addresses provide a handy abstraction for a cell location within the spreadsheet, whereas cell references play a larger role in TypeCalc.

For cell references, we allow the use of the R1C1 notation and A1 notation. The R1C1 notation can exist in two states; it can be relative, pointing to a cell relative to its own location, or absolute, pointing to an absolute cell location. Here, a relative reference is characterised by the use of brackets, which can be done on either rows or columns. An example of this would be =R[-3]C, pointing to the cell location three rows above its own cell location, or the example in Figure 24, which showcases the use of brackets on both rows and columns.

Figure 24: An example of a relative reference using the R1C1 notation. The importance of this figure is to highlight how R[-1]C[0] becomes a relative reference to the cell above, which has a green outline.

For A1 notation, we characterise relative and absolute references by the use of \$ symbols in the reference string. Cell references using A1 notation are relative if they do not use any \$ symbols. As such, if cell B2 contains the relative reference expression =A1, B2 will point directly to cell A1. But if cell B2 is copied and pasted to cell E2, the relative reference expression =A1 will change to =D1. However, if the reference was absolute =\$A\$1, then it will always remain the same when copied and pasted. Please refer to Figure 25 for an illustration between relative and absolute A1 notations. Notably, TypeCalc's implementation of this automatic adaptation of relative A1 notation references currently causes issues for absolute references written in R1C1 notation. When copied and pasted, TypeCalc will change the reference as if it was written in A1 notation, e.g., if copied down one row R1C1 becomes R2C2, and if copied one column to the right, it becomes S1D1.

	=A1			=D1
	=\$A\$1			=\$A\$1

Figure 25: Difference between relative and absolute A1 notations. The figure shows how copying a relative A1 reference will move the resulting expression by three columns, and how the absolute A1 reference will be maintained, even when moved.

In terms of classes associated with cell references, we have a similar setup as with cell addressing, meaning that for R1C1 notation, we use the R1C1RAREf class, while A1 references are created using A1RAREf. These classes inherit from a superclass SuperRAREf, which handles the operations. Our decision to make these specialisation classes are reflected upon in Section 12.1.1.

7.7 Undo/Redo Functionality

Our sixth quality goal was to implement undo/redo functionality to TypeCalc, which was not implemented in CoreCalc. We started brainstorming how to implement undo and redo, and at first glance, it seemed straightforward that when a user presses “CTRL+z”, the most recent action should be undone, and then, if “CTRL+y” is pressed, the action should be redone. These actions may be cached in some data structure like an array or a map. But then, more complex questions arose, such as: How do we cache actions? When should undone actions no longer be able to be redone? How do we undo and redo actions that involve cell duplicates in the cache? In this section, we will dive into these questions and explain how we implemented undo/redo in the back-end.

Initially in the development process, we cached a new SheetRep instance in an array for each new action made in the sheet. SheetRep is the class that represents the QT4 data structure. Our idea was that if “CTRL+z” or “CTRL+y” was pressed, we would change all the cells of that sheet to a step forward or backwards in the cache, and then recalculate the workbook to display the newly updated cells. However, we quickly realised that this was a very inefficient approach in terms of both time and space consumption, as it would require us to copy the entire sheet representation at every cell change. Therefore, we quickly abandoned this approach, instead choosing an idea that involved caching single cells as actions that could be undone and redone rather than whole sheets. In this way, we avoided caching a QT4 for each action, and we could get away with updating only one cell per undo/redo call. This solution is described in the following Sections 7.7.1 to 7.7.3 and reflected upon in Section 12.

7.7.1 Managing Undo/Redo History

We added three fields to the Sheet class to handle the undo/redo functionality: 1) `history` is an array with its type being an anonymous object type `{cell: Cell; row: number; col: number}` (Devero, 2021). Anonymous object types are used for defining an object without creating a specific type or interface for it. In this way, each element of `history` references a cell and its position in the sheet, and the purpose of `history` is to cache cell changes chronologically such that we can undo and redo them. 2) `historyPointer` is a field of type `number` that holds the index value of where the user is in the history. The purpose of `historyPointer` is to point to the position in the `history` where we should undo a cell. 3) The `undoCount` field is also of type `number`, and it is used to track how many actions have been undone. After declaring the three fields, the first essential step is to add cells to `history`. We do this inside `Sheet.Set()` because when a cell value is added to the sheet or updated, we also want to cache it in `history` to be able to undo it again. However, we set an upper limit for `history` of length 100 to avoid extensive space usage, so if more than 100 cells are added, the oldest cell in `history` is removed. Thus, one boundary is added here for when undone actions can no longer be redone.

There is also a second boundary for when undone actions can no longer be redone. That is, if a number of actions have been undone, and new actions are made in the spreadsheet, the undone actions will be removed from `history` and can no longer be redone. In short, this means that new actions overwrite undone actions. An example of this is visualised in the three steps of Figure 26. In step 1, three cells hold a value in the spreadsheet, and therefore, they are also cached in the `history` array. The most recently added cell is A3, indicated by the arrow. In step 2, CTRL+z is pressed to undo cell A3, and the most recent cell is now A2. A3 is no longer visualised in the

spreadsheet (as indicated with grey colour), but it is still cached in `history` in case it should be redone. In step 3, a new value is added in cell A4, and therefore, A3 has been removed from `history`, making A4 the most recent cell. It should be stated that we did not invent this boundary but were inspired by Excel’s undo/redone functionality, which seemingly works in the same way.

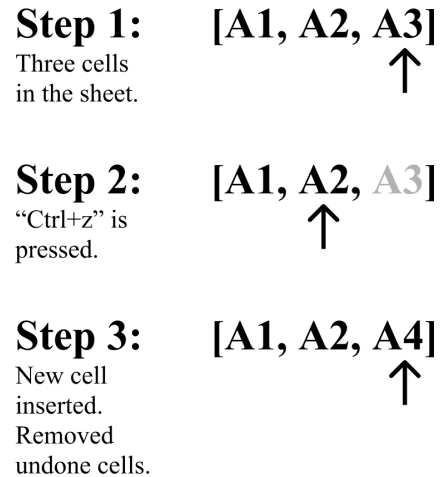


Figure 26: Visualisation of how undone actions are handled if new actions are made. They are deleted from `history` (see A3) and can no longer be redone. Note that the visualisation illustrates the concept and does not exactly match the code execution.

7.7.2 Sheet.undo()

`Sheet.undo()` is the method we implemented for undoing actions in `TypeCalc`. The functionality of the method will only be invoked if there are actions to undo, i.e. `undoCount` is smaller than the length of `history`. If this is true, `undoCount` is increased by one and `historyPointer` is decreased by one to indicate that we are pointing one step back in `history`. Then `undo()` checks if the cell already exists in `history` because in this case, it should rewrite the cell value to its previous state. Otherwise, if the cell only occurs once in `history`, `undo()` sets the cell to a `BlankCell` object. This functionality is illustrated in Figure 27. In step 1, two cells exist in `history`, and they are currently both visualised in the sheet. In step 2, cell A1 is updated to hold a new cell value, but the old version of A1 is still cached in `history`. In step 3, “CTRL+z” is pressed and `undo()` is invoked. `undo()` does not set A1 to be a `BlankCell` immediately, but searches through `history` to look for old A1 caches. `undo()` finds the old A1 instance and inserts it in the sheet.

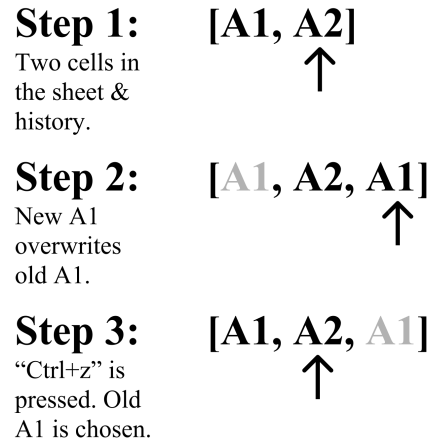


Figure 27: Visualisation of how `undo()` handles duplicate cells in the history. Note that the visualisation illustrates the concept and does not exactly match the code execution.

7.7.3 Sheet.redo()

`Sheet.redo()` is the method we implemented for redoing actions in TypeCalc. It is the simplest part of the whole undo/redo implementation. It checks if there are any actions to redo, meaning that `undoCount` is larger than zero. If so, `redo()` goes one step forward in history to redo an action. Unlike `undo()`, `redo()` does not have to search through history for duplicate cells, because it always has to take a single step forward, and if it encounters a duplicate cell in history, it can safely rewrite it. Lastly, `redo()` increments `historyPointer` by one and decreases `undoCount` by one. A call to `redo()` is shown in Figure 28. In step 1, two cells are seen in the sheet, but three cells are in history because A1 was updated. In step 2, “CTRL+z” is pressed to undo an action, such that the old A1 is shown again. In step 3, “CTRL+y” is pressed to redo() an action and the newest version of A1 is shown again.

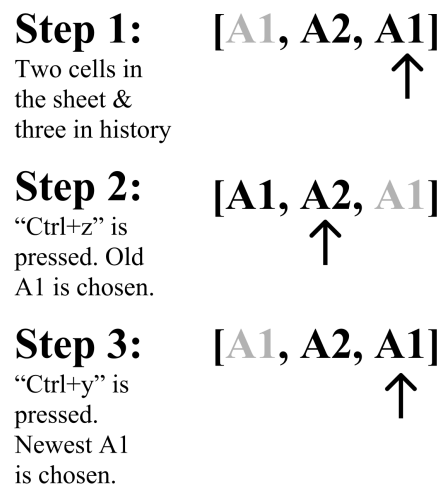


Figure 28: Visualisation of how `redo()` always goes one step forward in history if there are actions to redo. Note that the visualisation illustrates the concept and does not exactly match the code execution.

7.8 Recalculation

TypeCalc uses standard minimal recalculation (SMR), which tracks cycles and recalculates both volatile cells and edited cells as well as the cells they support (Sestoft, 2014b). The method that invokes SMR is called `Workbook.Recalculate()` and is seen in Figure 29. Inside `Recalculate()`, a call to `Workbook.TimeRecalculation()` is seen (line 2 in Figure 29), which takes an anonymous function as argument and returns the time it takes to compute this anonymous function. Anonymous functions are nameless functions defined inline where they are used, and therefore, cannot be referenced elsewhere in the code (Microsoft, 2025c). The advantage of this is that `TimeRecalculation()` can then be used for timing different recalculation operations, which can be used for benchmarking TypeCalc, as described in Section 5.5.

Cycle detection is handled differently during recalculation in TypeCalc than in CoreCalc. In CoreCalc, when a `CyclicException` is thrown by `Formula.Eval()`, `TimeRecalculation()` will catch it, aborting the current recalculation, and the next recalculation will initialise a full recalculation. Full recalculation is another type of recalculation used in CoreCalc to force recalculation of all active cells of the sheet, including unedited cells (Sestoft, 2014b). This full recalculation of CoreCalc will ensure that if other cells depend on a newly occurring cycle, these cells will be recalculated because their values are no longer valid. However, this results in the fact that not all cyclic cells will be discovered immediately in CoreCalc, but instead during the next recalculation. We did not find this solution optimal, because the next recalculation can occur in an indefinite amount of time when the user edits a cell, leaving some cyclic cells unmarked. Therefore, we instead designed `Recalculate()` of TypeCalc such that `CyclicExceptions` are not thrown and caught to avoid aborting the recalculation. Instead, we added a `cycleError` type to the `ErrorValue` class, such that when `Formula.Eval()` encounters a cycle, the value of the cell will be set to `cycleError`. Then, by letting the recalculation complete, other cells that depend on the cell with the `cycleError` will also obtain a `cycleError` through their expression evaluation. In this way, the user will immediately be presented with all the cells that are a part of a cyclic exception in the GUI, which we believe adds to a more transparent user experience, reflecting the present state of the cells. Furthermore, we believe that this software design decision negates the need for a full recalculation in case of cyclic dependencies, because SMR ensures that all edited cells and the cells they support will be recalculated, so no cells affected by a potential cycle will avoid recalculation.

In line 6 of Figure 29 the field `UseSupportSets` is set to `true` to indicate to `Formula.Eval()` that it should enqueue all the cells in its support set for evaluation - that is because, if a constant or formula cell has been evaluated, the cells that it supports should also be enqueued for evaluation. The boolean flag `idempotentForeach` is set to `true` (line 7, Figure 29), to avoid that cells which appear in multiple support ranges are enqueued for evaluation multiple times. Idempotence means that the result of a certain operation will remain the same after re-evaluation. An example could be if cell A1 supports area B1:C3 and cell A2 supports area B2:C4, then the cell area B2:C3 would be enqueued for evaluation twice if not handled properly. But CoreCalc and TypeCalc handle this in the `SupportArea` class with the method `foreachSupported()`, which checks the static `alreadyVisited` array for duplicate support areas. So when B2:C4 has been pushed once to `alreadyVisited`, other `SupportArea` objects will avoid applying the callback function passed to `foreachSupported()` on the overlapping support area B2:C3. The callback function will be either `MarkCellDirty()` or `EnqueueCellForEvaluation()`. Thus, the implementation of

idempotence avoids redundant evaluation of cell areas that exist in multiple support sets.

The next step of Recalculate is to mark all edited and volatile cells Dirty (line 5-10, Figure 29), which will also recursively mark cells in the associated support sets Dirty (Sestoft, 2014b). Marking a cell Dirty means invalidating its cached value, and thus, signalling that the cell should be recalculated. The purpose of also marking cells of support sets Dirty is to ensure that a cell that depends on a recalculated cell will also be recalculated itself. Then, we clear the awaitsEvaluation queue, which was done in CoreCalc to empty the queue in case a recalculation was aborted (line 11, Figure 29). We decided to let the clearing process stay for the sake of safety in case a recalculation of TypeCalc would be unexpectedly aborted. In lines 13-21 (Figure 29), we enqueue edited and volatile cells for evaluation, which will add all Dirty cells to the awaitsEvaluation queue. We are not entirely sure whether it is necessary to both mark cells Dirty and enqueue them for evaluation, because it seems like the Dirty state might be redundant in the code. However, this is a theory that appeared to us late in the project, and we did not manage to look further into it. Therefore, in future work with TypeCalc, this is something we would look into. Lastly, in lines 19-21 (Figure 29), the enqueued cells are being evaluated one by one, starting from the beginning of the queue.

```

1      public Recalculate(): number {
2          return this.TimeRecalculation(() => {
3              this.UseSupportSets = true;
4              SupportArea.idempotentForeach = true;
5              this.volatileCells.forEach((fca: FullCellAddress) => {
6                  Cell.MarkCellDirty(fca.sheet, fca.cellAddress.col,
6                      ↪ fca.cellAddress.row);
7              });
8              this.editedCells.forEach((fca: FullCellAddress) => {
9                  Cell.MarkCellDirty(fca.sheet, fca.cellAddress.col,
9                      ↪ fca.cellAddress.row);
10             });
11             this.Clear("awaitsEvaluation");
12             SupportArea.idempotentForeach = true;
13             this.volatileCells.forEach((fca: FullCellAddress) => {
14                 Cell.EnqueueCellForEvaluation(fca.sheet, fca.cellAddress.col,
14                     ↪ fca.cellAddress.row);
15             });
16             this.editedCells.forEach((fca: FullCellAddress) => {
17                 Cell.EnqueueCellForEvaluation(fca.sheet, fca.cellAddress.col,
17                     ↪ fca.cellAddress.row);
18             });
19             while (this.awaitsEvaluation.length > 0) {
20                 this.awaitsEvaluation.shift()?.Eval()
21             }
22         });
23     }

```

Figure 29: Extract from Workbook.Recalculation() which executes SMR in TypeCalc.

We encountered some problems with recalculation for specific long dependency chains. We will elaborate on this in Section 12.1.3.

8 Function Evaluation

In continuation of the back-end section, we want to explain Formula.js and TypeCalc’s function implementation. This includes the Function type, the methods `FunCall.Make()` and `FunCall.Eval()`, our `ExprArray` class, and non-strict functions.

8.1 Functions and Formula.js

One of our most significant implementation decisions was to replace the original function implementation in CoreCalc with the open-source Formula.js library (Ghalimi et al., 2025a). Formula.js is a community-built implementation of most of the built-in functions from Excel. It is extensively tested and, in most cases, similar to the functions in Excel and CoreCalc, with minor deviations regarding naming conventions and argument handling. The name of Formula.js is a bit misleading as it only implements functions and not formulas (distinction is described in Section 2.2.2). At the time of implementing Formula.js to our project, we had three main reasons for doing so: 1) It spared us from translating roughly 1200 lines of code from the `Functions.cs` file in CoreCalc. 2) The Formula.js library contains a larger number of implemented functions and is updated semi-regularly, and 3) since Formula.js is written in JavaScript, it was already compatible with our back-end and could quickly be imported and implemented. However, arguments 1 and 3 were later found to be partially inaccurate, which we will elaborate on in Section 12.1.7. By implementing Formula.js, we had to rewrite some of the functionality of the `FunCall` class in `Expressions.ts`. We also had to declare a new type for functions and develop a new class called `ExprArray`, which we will describe in the following sections.

8.2 New Function Type

In CoreCalc, a function has type `Function`, which is a class declared in `Functions.cs`. Since we replaced `Functions.cs` with Formula.js, we had to define our own type for functions, which is seen in Figure 30. It is called `functionType` and is declared with the `type` keyword used for making custom types in TypeScript. It represents a TypeCalc function which takes a number of arguments and returns a result. The arguments are declared as `...args`, which is of type `array`. The three dots in front of `args` make it a rest parameter, meaning it is treated as having indefinite length (Microsoft, 2025c). The parameter `...args` can hold a union of six different types, which are the types we deemed relevant for handling function arguments in TypeCalc (line 1, Figure 30). The `nestedArgs` type is the most noteworthy of the six types. It is a custom type used for handling nested arguments, i.e. arguments that come in the form of an array. This type of argument is a special form of expression called an expression array, which is elaborated in 8.4. The return type of `functionType` is also a union of types, but not necessarily an array. That is because most functions in Formula.js return a single result and only a few functions return an array of numerical results. There are some variations between the `...args` types and the return types, which are `Date`, `string[]`, and `boolean`. One reason for this is, for example, that we do not need `Date` as an argument type because in the GUI of TypeCalc, dates will be written as strings such as “3/15/11”, which the `Date` functions of Formula.js correctly handle. However, the `Date` functions of Formula.js return `Date` objects, so therefore, `functionType` has `Date` object as one of the return types.

```

1  type functionType = (...args: (string | number | boolean | ErrorValue | null |
   ↪ nestedArgs)[])
2    => string | number | boolean | ErrorValue | Date | number[];
3
4  type nestedArgs = (string | number | boolean | ErrorValue | null |
   ↪ nestedArgs)[]

```

Figure 30: Extract from Expressions.ts where we added a custom type called `functionType`. The purpose of `functionType` is to replace the previous `Function` type of `CoreCalc`. The custom type called `nestedArgs` is used to handle nested arguments in case an expression array occurs in a formula cell 8.4.

8.3 FunCall.Make()

The purpose of the static `Make()` method in `FunCall` is to return a new instance of `FunCall` that holds a reference to the function it should invoke and an array of the expressions it should invoke this function on. First, `Make()` checks if the function name is similar to any of the functions we defined ourselves. Since `Formula.js` does not include functions for arithmetic operators, e.g. “+”, “-”, and “/”, we implemented the functions `ADD(...)`, `SUB(...)`, and `DIVIDE(...)` ourselves to enable the user to use basic arithmetic expressions (lines 3-11 in Figure 31). If the function in `Make()` does not match with any of our functions, then it calls a helper method `getFunctionByName()` to retrieve the desired function from `Formula.js` (line 13 in Figure 31). The helper method checks if the function exists in the library and then casts it to our `functionType` and returns it. Lastly, `Make()` ensures that neither the function nor the expressions are null or undefined (lines 14-22 in Figure 31) and returns a new instance of `FunCall` (line 23 in Figure 31).

```

1  public static Make(name: string, es: Expr[]): Expr {
2    ...
3    if (name === "DIVIDE") {
4      return this.DIVIDE(es);
5    }
6    if (name === "SUB") {
7      return this.SUB(es);
8    }
9    if (name === "ADD") {
10     return this.ADD(es);
11   }
12   ...
13   const func: functionType | null = FunCall.getFunctionByName(name);
14   if (func === null) {
15     return new ErrorConst(ErrorValue.nameError);
16   }
17
18   for (let i = 0; i < es.length; i++) {
19     if (es[i] === null || es[i] === undefined) {
20       es[i] = new ErrorConst(ErrorValue.valueError);
21     }
22   }
23   return new FunCall(func, es);
24 }

```

Figure 31

8.4 Expression Arrays

As a supplement to CoreCalc's range of expression classes, we decided to implement a new class called `ExprArray`. The reason for this was that some functions in `Formula.js`, such as `SUMIF`, `FREQUENCY`, and `AGGREGATE`, accept a combination of scalar and array arguments. When an array argument is written in a function call, for instance, `SUMIF([1, 2, 3, 4], ">2")`, we need to detect the beginning and end of the array, extract its values, and handle the last string argument. `Formula.js` cannot handle the arguments if they are all provided as scalar arguments, e.g. `SUMIF(1, 2, 3, 4, ">2")` without square brackets, so we had to accommodate for this. Therefore, we came up with the idea of expression arrays such that array arguments would be of type `ExprArray`, which can hold zero or more expressions in an array. When we evaluate function calls, we use a helper method `getExprValues()` to compute the values from each expression (Figure 32). The `map` method applies a mapping function to each expression (line 4, Figure 32), such that we check the expression's type and handle it accordingly. If an expression is of type `ExprArray`, we recursively call `getExprValues()` to extract each value of the expression array and return a new array with the contents of that expression array (line 6, Figure 32). Lastly, `getExprValues()` returns one array containing all the values of each expression, including nested arrays of type `nestedArgs` if expression arrays occurred, which is used by `FunCall.Eval()` to provide the arguments for the `Formula.js` functions as described in 8.6.

```

1      public static getExprValues(sheet: Sheet, col: number, row: number, es:
      ↪ Expr[]):
2          (string | number | boolean | ErrorValue | null | nestedArgs)[] {
3
4          return es.map(expr => {
5
6              if (expr instanceof ExprArray) {
7                  return FunCall.getExprValues(sheet, col, row,
      ↪ expr.GetExprArray());
8              }
9              const value: Value = expr.Eval(sheet, col, row);
10
11              if (value instanceof ErrorValue) {
12                  return value;
13              }
14              if (value instanceof NumberValue) {
15                  return NumberValue.ToNumber(value)
16              }
17              ...
18          }

```

Figure 32: Extract from the helper method `FunCall.getExprValues()` used by `FunCall.Eval()`. If an expression array of type `ExprArray` is encountered as in line 6, `FunCall.getExprValues()` is called recursively to extract the contents of the expression array.

8.5 Handling Non-Strict Functions

TypeCalc implements the non-strict functions `IF(...)` and `CHOOSE(...)`. As mentioned in Section 2.2.2, non-strict functions do not evaluate all of their arguments but only the arguments they need. When evaluating non-strict functions, we start by checking the boolean flag `nonStrict`, which is a field in `FunCall`, and it will only be true for `IF` and `CHOOSE`. We then evaluate only the

needed expression. For example, with the formula `=SUM(CHOOSE(2, A1:A10, B1:B10))` we sum only the evaluation of the argument at the third index, i.e. `B1:B10`, and we never evaluate `A1:A10` in this case because of the non-strictness of the function. We handle `CHOOSE(...)` by evaluating the first argument, which is "2" in this case, and then we evaluate the argument at position 2 (Figure 33). This also means that if the formula was inserted at cell A1, for example, and the argument A1:A10 therefore caused a cyclic exception, we deliberately do not throw an error because the argument is never evaluated. This is in accordance with rules for non-strict functions specified in Section 2.2.1.

```
...
return this.es[(this.es[0].Eval(sheet,col,row).ToObject() as
↪ number)].Eval(sheet,col,row);
...
```

Figure 33: Extract from `FunCall.Eval()` where the non-strict `CHOOSE(...)` function is handled.

8.6 *FunCall.Eval()*

The `FunCall.Eval()` method was introduced with some changes to fit with the `Formula.js` library and our function implementations. Its purpose is to evaluate the result of the function call on the provided arguments. Firstly, the method handles non-strict functions as described in Section 8.5. Secondly, `getExprValues()` is called as described in Section 8.4, and the result is stored in an array called `args`. Then the function, originating from either `Formula.js` or our function implementations, is invoked on the arguments and the result is stored in the variable `result`. If `result` is of type `Date`, `String`, or `Boolean`, a new `TextValue` containing the result is returned, and if it is of type `Number`, a `NumberValue` is returned. However, if the result is of type `Array`, we are dealing with an array-valued result that should be displayed in a cell area. Therefore, we return a new instance of `ArrayExplicit`, which is used for caching the result of a cell area (line 12, Figure 34). Lastly, if the function is not implemented, `FunCall.Eval()` returns an `ErrorValue`. In `TypeCalc`, only the functions called `FREQUENCY(...)` and `MODEMULT(...)` (described in 2.2.2) output array-valued results for a cell area, and since they both output their result to a one-dimensional cell area, we decided to add all of their values in a column-based manner only (line 3, Figure 34). The `FREQUENCY` function in `Formula.js` calculates how many values occur within a given range. For example, `=FREQUENCY([1,2,3],[2])` returns `[2,1]` because two values are `<= 2` and one value is `>2`.

```
1  ...
2      if (Array.isArray(result)) {
3          const values: Value[][] = [[]]; // Works for a column-oriented
4          ↪ result.
5
6          for (let i = 0; i < result.length; i++) {
7              values[0][i] = NumberValue.Make(result[i]);
8          }
9
10         const start = new SuperCellAddress(0, 0);
11         const end = new SuperCellAddress(0, result.length - 1);
12
13         return new ArrayExplicit(start, end, values);
14     }
```

Figure 34: Extract from FunCall.Eval() where a result of type Array is handled. The results of the functions FREQUENCY and MODEMULT will match this if statement, and the result will be cached in an instance of ArrayExplicit.

9 Parsing

To round out our walk-through of the back-end related areas, we also provide a thorough explanation of our parsing of input. This section will explain our parsing logic, the libraries we considered using, the classes our parser consists of, and the challenges we had working in TypeScript with our chosen library.

9.1 Parser Logic

As part of our spreadsheet implementation, we have written our own parsing logic, such that a user can click on a cell and insert a textual input (e.g. `=SUM(10,10)`). However, to ensure that our spreadsheet implementation can identify the different elements of some free-form text written by us in a cell, we needed to present it with some internal representation of what different characters and words mean. To define a parser, we have to complete two tasks: 1) define a set of lexical tokens, and 2) create rules for how to act when encountering a lexical token. To define lexical tokens, we make use of *grammars*, which are sets of rules for combining symbols to well-formed text (Sestoft, 1999). A grammar G is defined by a set of terminal characters, a set of non-terminal characters, a set of rules, and a starting symbol belonging to the non-terminals. A lexical token is then formed by starting from a non-terminal character, matching it to a rule, and then proceeding to match these characters unless all characters result in a terminal character. If this is the case, then we have formed a lexical token (Sestoft, 1999). For instance, if we wanted to recognise a number token, from the text input “10A” we would look repeatedly for numbers in a row, continuing until reaching the end of characters, or encountering a letter. As for the parsing, the parser has a two-fold purpose: 1) It needs to verify that the input that we have provided follows the grammar rules defined, and 2) it should help us understand how the grammar rules were applied and which alternatives were used when generating the syntax tree (Sestoft, 1999). We generally distinguish between two types of syntax trees: *Concrete Syntax Tree* and *Abstract Syntax Tree*, where the former is a full representation of the syntactical structure, which includes all lexical tokens found during the parsing step. The latter is a more simplified version of the concrete syntax tree, that focuses more on the semantic representation rather than the syntactical details (Chevrotain, 2025b).

9.2 Purpose and Library Considerations

Our implementation of the grammar rules and parsing logic has been done using the *Chevrotain* library; a popular LL(k) parser building toolkit made for JavaScript, which can generate a Concrete Syntax Tree (Chevrotain, 2025a) based on grammar rules and parsing rules. Being an LL(k) parser means that Chevrotain can process restricted context-free grammar moving from left-to-right, using the left-most derivations and being able to look k tokens ahead, until it matches a grammar rule. Our choice of parsing logic framework was based on a number of considerations. In the beginning, our goal was to stick as close to the CoCo/R version that was used for CoreCalc (Sestoft, 2014b). This introduced problems with converting the ATG file to fit the CoCo/R port (Mingodad, 2022). This led us to try another parsing framework ANTLR (Parr, 2025), which appeared to resemble CoCo/R. This, however, was very hard to comprehend, and given our timeframe for the project and very limited experience with creating parsing logic, we looked into more finished solutions, such as the Hyperformula parser (Handsontable, 2025b) or the Fastformula parser (Lyu, 2024). The benefit of using this type of parser, which was already implemented, was that we would have a finished

product to work with directly, since both of these parsers had integrated with the Formula.js library. While this would have significantly reduced the time spent on constructing the parser, the fear of having to introduce much unnecessary complexity due to overwriting certain parts of this solution motivated us towards rewriting the parser specification ourselves. Finally, we also considered parsing efficiency, and based on a benchmark from Chevrotain themselves, we found that using Chevrotain beats all of the existing JavaScript parser implementations (Chevrotain, 2025c). We have since learned that this claim is only true on chromium-based browsers.

Our implementation is based on the Spreadsheet.ATG file, which is part of the CoreCalc implementation. In total, our parsing implementation ended up consisting of three classes: SpreadsheetLexer, SpreadsheetParser, and SpreadsheetVisitor. The separation of these classes was helpful in managing the parsing logic and created a clear connection to both the grammar part and the parsing rule part of TypeCalc. In the following subsections, our implementation of these classes will be addressed.

9.3 SpreadsheetLexer

The SpreadsheetLexer class is in charge of producing the grammar step for our parsing logic; It defines what characterises a lexical token. In Chevrotain, we define tokens using regular expressions (henceforth regex) originating from the ECMAScript language, which defines the regex syntax used in JavaScript and TypeScript. Our lexical tokens were created using the `createToken()` method, which takes in a name, a regex pattern, and additional parameters, such as tokens to skip, if needed. An example of our definition for the WhiteSpace token and Number token can be seen in Figure 35. After defining all the lexical tokens, we added all of these to the static field `AllTokens`, allowing them to be used outside of the class.

```
class SpreadsheetLexer {
  static WhiteSpace: TokenType = createToken({
    name: "WhiteSpace",
    pattern: /\s+/,
    group: Lexer.SKIPPED });
  static NUMBER: TokenType = createToken(
  {
    name: "Number",
    pattern: /\d+(\.\d+)?([eE][+-]?\d+)?/ );
}
```

Figure 35: SpreadsheetLexer class definition with token types.

In our lexical token definitions, we had to account for the scenario where a piece of text could be matched by multiple tokens. An example of this is the overlap of tokens when used for the R1C1 notation. A complete overview of the cell reference notations used for the parser can be seen in Figure 36, where the Raref token created for A1 references is also included. The problem occurs when the grammar rules overlap, such as in the case with XMLSSRaref11 and XMLSSRaref12. Here, the string input `RC[9]`, which is meant to be read as XMLSSRaref12, is recognised as a XMLSSRaref11 token. To solve this, we changed the ordering of the grammar rules, setting XMLSSRaref12 before XMLSSRaref11. This, in turn, tells the lexer to evaluate the token definitions in a certain order, giving precedence to those that appear first in the `AllTokens` list.

```
createToken({name: "A1Ref", pattern: /\$?[A-Z]+\$?[0-9]+|\$?[A-Z]+[0-9]+/});
createToken({name: "XMLSSRRef11", pattern: /RC/});
createToken({name: "XMLSSRRef12", pattern: /RC[0-9]+/});
createToken({name: "XMLSSRRef13", pattern: /RC\[+-]?[0-9]+/});
createToken({name: "XMLSSRRef21", pattern: /R[0-9]+C/});
createToken({name: "XMLSSRRef22", pattern: /R[0-9]+C[0-9]+/});
createToken({name: "XMLSSRRef23", pattern: /R[0-9]+C\[+-]?[0-9]+/});
createToken({name: "XMLSSRRef31", pattern: /R\[+-]?[0-9]+C/});
createToken({name: "XMLSSRRef32", pattern: /R\[+-]?[0-9]+C[0-9]+/});
createToken({name: "XMLSSRRef33", pattern: /R\[+-]?[0-9]+C\[+-]?[0-9]+/});
```

Figure 36: A complete overview of the cell reference types defined as lexical token definitions. Tokens are generally divided into groups, such as 11-13, 21-23, and 31-33

9.4 SpreadsheetParser

Parsing rules in Extended Backus-Naur Form

Expression	<code>:= logicalTerm {logicalOp logicalTerm}</code>
raref	<code>:= A1Ref XmlssRAREf11 XmlssRAREf12 XmlssRAREf13 XmlssRAREf21 XmlssRAREf22 XmlssRAREf23 XmlssRAREf31 Xmlss- RAREf32 XmlssRAREf33</code>
exprs1	<code>:= expression {(Comma Semicolon) expression}</code>
factor	<code>:= application Minus number number [SheetRef] raref [Colon raref] LBracket [factor {Comma factor}] RBracket True False StringLiteral LParen expression RParen</code>
cellContents	<code>:= Equals expression QuoteCell Datetime StringLiteral number Minus number TRUE FALSE</code>
term	<code>:= powFactor {mulOp powFactor}</code>
mulOp	<code>:= Multiply Divide</code>
logicalOp	<code>:= Equals NotEqual LessThan GreaterThan LessThanOrEqual GreaterThanOrEqual</code>
addOp	<code>:= Plus Minus Ampersand</code>
powFactor	<code>:= factor {Power factor}</code>
logicalTerm	<code>:= term {addOp term}</code>
application	<code>:= Identifier LParen [exprs1] RParen</code>
number	<code>:= Number</code>

Table 3: This table visualises how each parsing rule works through the use of the EBNF syntax. The number rule consumes a Number token, meaning that it does not perform infinite recursion. To better separate tokens from rules, token are capitalised, whereas rules are in full lowercase.

With our lexical tokens defined, we constructed the SpreadsheetParser class based on the CST-parser class from Chevrotain. The purpose of SpreadsheetParser is to generate the CST, while

also determining if a provided input text is valid with respect to the lexical tokens that have been defined. In Chevrotain, a CST is created by defining a set of parsing rules, which determine how to build the tree structure. Since our parsing logic is based on the CoreCalc implementation, we start CST creation at the cellContents rule. Here we determine if the input contains a formula or a constant. If it is a constant, we end the CST there, with only a single leaf node. Otherwise, we keep building the CST structure until reaching the end of the input text or until we can match no more tokens. To exemplify this, consider the input text `"= 10 + 10"`. Starting from the cellContents rule, we immediately recognise the first pattern `"="` Expression, prompting us to follow the expression subrule. Here, we would add the `"+"` to the CST, and follow the `"10"`s through multiple subrules until reaching the factor subrule, which consumes the `"10"` token, finalising the CST, since there are no more tokens to match. To better visualise the relation between rules and subrules in Chevrotain, we include a table on all rules, depicted in EBNF form (see Table 3). Chevrotain has a basic syntax for writing the parser class, using `$.RULE` to define a parser rule. Within a `$.RULE`, we can process a part of the input using the `$.CONSUME` function. If there are alternative possibilities, we use the `$.OR` keyword, allowing a single token found within any of the alternating branches to be chosen. We can combine a `$.CONSUME` call with a `$.SUBRULE` call, which will process the remaining input, excluding the consumed token, in another parsed rule.

In several cases, we are interested in being able to parse more than singular expressions, such as `"= 10 + 10 + 10 * 10"`, and this is done using the `$.MANY` keyword in the parser, that tells the parser to look for zero or more of the pattern described within the `$.MANY` keyword. The use of the `$.MANY` keyword is apparent in Figure 37, where an expression can consist of more than two logical terms and a logical operator. Likewise, the use of `$.OPTION` signifies that a rule is not required for the parsing tree, and can be skipped if it is not found during the CST creation. This can be seen in Figure 38.

```
function expression() {
  $.SUBRULE($.logicalTerm);
  $.MANY(() => {
    $.SUBRULE2($.logicalOp, { LABEL: "Operator" });
    $.SUBRULE2($.logicalTerm);
  });
}
```

Figure 37: Example of the "expression" rule in SpreadsheetParser class using Chevrotain Parser pattern. This rule follows a single logicalTerm, and then zero or more logical Operators and additional logicalTerms. If a Chevrotain subrule uses the same rule pattern more than once, within a single rule, it must use `$.SUBRULE2`, `$.SUBRULE3`, and so on.

While most of the parser was a port of the specification file from CoreCalc, we had to expand our parser such that it would recognise more advanced formulas, which were part of the Formula.js library. This resulted in the introduction of square brackets in our lexer, pattern matching that looks for `"["` followed by zero or more calls to the expression rule, and then finally a `"]"`. The idea here was to generate an array-type, which could be used in the same manner as other values, which can be seen in Figure 39. In addition to this, we also encountered a problem with formulas that required boolean values, which also prompted us to add a way of recognising `true` and `false` values in formulas. Up until this point, we had issues differentiating between functions and boolean values, since the identifier token was prioritised higher than our boolean values. Through an addition to


```
function Factor() {
    .... Other rules and syntax
    ALT: () => {
        $.OPTION3(() => {
            $.CONSUME(SpreadsheetLexer.SheetRef);
        });
        $.SUBRULE($.raref)
        $.OPTION4(() => {
            $.CONSUME(SpreadsheetLexer.Colon);
            $.SUBRULE2($.raref);
        });
    },
}
```

Figure 38: Snippet of factor rule in SpreadSheetParser. In factor, a sheet reference may be provided by writing "sheetName!", which can be followed by a raref. However, the parsed input is not required to include a sheet reference.

the lexer and parser, which recognised true and false, it worked with boolean values.

```
function Factor() {
    .... Other rules and syntax
    {
        $.CONSUME(SpreadsheetLexer.LBracket);
        $.OPTION2(() => {
            $.SUBRULE2($.expression, { LABEL: "ArrayElement" });
            $.MANY(() => {
                $.CONSUME(SpreadsheetLexer.Comma);
                $.SUBRULE3($.expression, { LABEL: "ArrayElement"
                ↪ });
            });
        });
        $.CONSUME(SpreadsheetLexer.RBracket);
    }
}
```

Figure 39: Snippet of factor rule in SpreadSheetParser. The Snippet showcases our extension to the parser, allowing for more advanced formulas in Formula.js to be evaluated.

9.5 SpreadsheetVisitor

The final layer of our parsing logic was the SpreadsheetVisitor class, which traverses the CST generated by the lexer and parser classes. The purpose of the visitor class was to integrate CST with the rest of our codebase by applying semantics to the CST while it is being traversed. In Chevrotain, to use a visitor class, we must define each parsing rule and provide semantics on how to act when encountering either a token or subrule. As such, all the rules defined for the parser received a namesake equivalent in the visitor class. To visit the different nodes in the syntax tree, we make use of the Chevrotain visitor pattern, using the `this.visit()` method to look at the children of the specified node. The `visit()` method can be thought of as a way of traversing the syntax tree and looking into different layers.

The visitor class begins at the root of our previously generated CST, following the child nodes according to the CST in a depth-first manner. For instance, if we had generated a syntax tree on the input `"= 10 * 10"`, it would initially follow the `"="`, and continuing with the remaining `"10 * 10"` in the next level of the tree, which in this case would be one or more logical terms, then

terms, and so on. At the bottom of the tree it would encounter two leaf nodes containing the 10's, labelling these as NumberValues, and returning these values upwards in the parsing tree, until reaching the term level again, where it would process the NumberValues and the "*" symbol as a FunCall, returning it as "MULTIPLY, 10, 10", until reaching the root again, and finally returning "= MULTIPLY, 10, 10". Since we encountered an "=" token in the cellContents rule, we know that upon returning the FunCall expression to cellContents, we should return a Formula cell as seen in Figure 40. Should we not encounter a "=" token in the CST, we check for other matching tokens, and return these according to their matched type, as seen in the Figure 41

The way that our parser and visitor were implemented was also helpful in solving another fundamental problem, namely, ensuring that the arithmetic ordering of our input was being upheld at all times. To solve this issue, we leveraged the fact that our parsing order had an existing hierarchical nature when it came to evaluating the arithmetic precedence order. Since exponents and square roots would be evaluated first in the lowest levels of the tree, and have their result sent upwards to a lower priority operator, we would always ensure that the correct evaluation of values with different arithmetic would be done. In certain cases, we also had to account for the scenario where parentheses were used, and here we created a call to look at another expression within the factor() method, such that it would evaluate the expression at a lower node before it would return the value upwards. This would ensure that the inner expression of the parentheses would be evaluated before all other outer expressions.

9.5.1 Integration with Codebase

To incorporate the whole lexing, parsing, and visiting into the codebase, we created a method called Parse(), which processes a string, a workbook, and their respective column and row values for the cell. In turn, it returns the Cell type from the visitor cell field, which is assigned in the cellContents() method. As we integrated the Parse() logic, we discovered a problem with connecting Formula.js and our parsing logic. This was related to the fact that Formula.js does not implement simple arithmetic functions, such as addition, subtraction, and division, and instead relies on the fact that JavaScript itself has functionality for this. This required us to apply additional logic to many of our operators, renaming "+" to "ADD", "-" to "SUB", as well as creating additional methods for FunCall, which is covered in Section 8.1.

```
protected cellContents(ctx: CellContentsCstChildren): Cell {
  const e:any = this.visit(ctx.expression);
  if (ctx.Equals) {
    this.cell = Formula.Make(this.workbook, e!);
  }
  // ... more code.
  return this.cell;
}
```

Figure 40: Snippet of cellContents rule in SpreadSheetVisitor. The snippet showcases both the initial step for our visitor class as well as where the final result is returned as a cell.

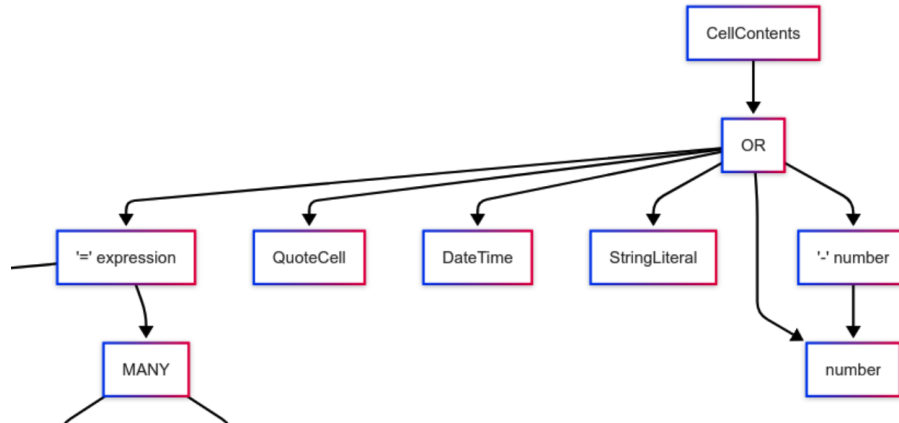


Figure 41: The Starting point of the Visitor class. Depending on the input that is parsed, the value at the associated node will return a `cellObject`.

9.6 TypeScript and Chevrotain

As previously mentioned, Chevrotain is a parser toolkit developed for JavaScript. Since TypeScript is a superset of JavaScript, this means that Chevrotain is compatible with TypeScript as well. One problem that we encountered during this process is the fact that Chevrotain relies on JavaScript, which uses dynamic type inference (Mozilla Developer Network, 2025a) and this behaviour is translated into TypeScript with the use of the *any* type. The use of the *any* type is generally considered bad practice, since it removes our ability to use type safety at compile time, in the same way that C# does (Microsoft, 2025d). Looking at the documentation, we found that there are no problems with recognising our lexical tokens in TypeCalc, since they are defined using the *TokenType* interface in Chevrotain. When it came to the parser class, we had several problems with registering the correct type inference for each of the parsing rule definitions. While this issue could technically be solved using a multitude of interfaces, we decided to forego this edit for our initial version of TypeCalc. The reasoning behind this was partially time constraints, but also the fact that our parser worked as intended. Since types are erased when TypeScript is transpiled to JavaScript, our code would still be working. Finally, for the visitor class, we initially experienced the same problem with the *any* type being used for method parameters, which was in line with Chevrotain’s documentation on visitor methods. However, through the use of the `generateCstDtsopen()` method in the `CstParser` class, we could generate a series of interfaces, allowing for type safety and easier access to syntax tree nodes.

10 Front-end

The final part of the TypeCalc implementation is the front-end that is responsible for creating and managing the GUI. In the following section, we will explain how we arrived at our chosen grid implementation and provide an overview of the various parts that constitute the front-end, elaborating further on our React components along the way. We will also give a rundown of the features we prioritised implementing, going in-depth with the more complex ones.

10.1 Choice of Grid

Initially, we experimented with a regular HTML table as the grid of TypeCalc (Mozilla, 2025b). One of the key challenges we quickly identified with translating a spreadsheet to a browser was the need for a spreadsheet to be able to have billions of cells. Expectedly, it is extremely inefficient to render every cell all the time. This is especially true for a browser application, where it has to compete for resources with the browser itself and typically also other websites that the user may have open. This was also the issue of the HTML table, for though it was able to be formatted as desired and was easy to work with, it does render every cell, even if empty or off-screen. As a result, the HTML table scaled up poorly and was deemed unsuitable. This led us to look into existing grid implementations that utilised UI virtualisation, meaning we only render what is actually currently visible on-screen (Chinnathambi, 2020). Given the aforementioned inefficiency of rendering all cells all the time, it follows that the ability to limit ourselves to only rendering visible cells could prove essential in keeping TypeCalc responsive and fast. In our search, we discovered the *react-virtualized* library, which has components that fulfilled this criterion, but after considering which functions we needed from the grid itself, we decided to follow the advice of the library’s author to instead use *react-window* (Vaughn, 2025a). *react-window* is another library by the same author that sacrifices features in exchange for taking up less storage and having both better performance and documentation. This was a trade-off that suited our needs, and so we ended up using *react-window*’s `VariableSizeGrid` component for TypeCalc’s grid. A further consequence of this was that it made it an easy choice to build our front-end with the React library (React, 2025c).

10.2 Overview of Front-end Files

A key goal of our project was to provide a GUI for TypeCalc. Given the popularity of the spreadsheet applications Excel and Google Sheets, we looked to their construction for inspiration in order to try to maximise external consistency (Nielsen, 1994). What this means is that our choices should follow established industry conventions as much as possible so that users do not “have to wonder whether different words, situations, or actions mean the same thing” (Nielsen, 1994, Section “4: Consistency and Standards”). The GUI is built up around an HTML file (`index.html`), a CSS style sheet (`style.css`) and several TypeScript files (see Figure 15). The primary of these is `SpreadsheetGrid.tsx`, which contains the majority of the code used to define the visual representation of the spreadsheet in the React component `VirtualizedGrid` (React, 2025a). It has three custom subcomponents (children) of its own, `SheetHeader`, `SheetFooter` and `GridCell`, which are in separate TypeScript files of the same name. Children can be passed prop-objects (props) from their parent component, containing information such as variables or functions that the child needs for its functionality (React, 2025b). In TypeCalc, `VirtualizedGrid` passes several props to two of its children - `SheetFooter` and `GridCell`. The spreadsheet defined by all these compo-

nents is rendered in `CreateGrid.tsx`, where we also specify the number of columns and rows of the sheet. The last TypeScript file, `HelperFunctions.tsx`, contains various helper functions used by our components.

10.2.1 VirtualizedGrid Component

```

1  export const VirtualizedGrid: React.FC<GridProps> = ({
2    columnCount,
3    rowCount,
4    columnWidth = 80,
5    rowHeight = 30,
6    colHeaderHeight = 40,
7    rowHeaderWidth = 40,
8    width = window.innerWidth,
9    height = window.innerHeight * 0.92,
10 }): GridProps => {
11   const colHeaderRef = useRef<Grid>(null);
12   const rowHeaderRef = useRef<Grid>(null);
13   const bodyRef = useRef<Grid>(null);
14   const [scrollOffset, setScrollOffset] = useState({ left: 0, top: 0 });
15   const [sheetNames, setSheetNames] = useState<string[]>(["Sheet1"]);
16   const [activeSheet, setActiveSheet] = useState(sheetNames[0]);
17   const [windowDimensions, setWindowDimensions] = useState({
18     width: window.innerWidth,
19     height: window.innerHeight,
20   });

```

Figure 42: VirtualizedGrid’s signature and fields.

VirtualizedGrid is the primary React component of TypeCalc’s front-end. It builds on the VariableSizeGrid component from `react-window`, having many of the same props used to define the dimensions of various parts of the grid, such as `columnWidth` and `rowHeight`, but also a few additions of our own design like `colHeaderHeight` and `rowHeaderWidth`, as seen in Figure 42 (Vaughn, 2025b). Originally, we defined our props in this manner to easily pass them to the various instances of VariableSizeGrid that we create while also allowing us to resize the sheet based on user input for the props `columnCount` and `rowCount`. Eventually, however, settled on having the sheet always be full-size since it comes with no performance cost thanks to the virtualisation. Also of note are the VariableSizeGrid props `overscanRowCount` and `overscanColumnCount`, which are used to render rows and columns outside of the viewport. In our case, we render an additional five rows and columns, as opposed to the default of one row and column, in order to make fast scrolling appear smoother, as is shown in Figure 43. VirtualizedGrid has four state variables, each with an appropriate state setter function that are managed by the React hook `useState` (React, 2025g). State variables differ from regular variables, as they need to remember their current values even as parts of a page re-render due to user interactions. The state setter function then is used to change its corresponding state variable, and trigger a re-render when doing so - something that changing a regular variable would not trigger. For example, if you create a new sheet and switch to it, you need to re-render to properly reflect which sheet is active. If you then create another sheet, you still want the `activeSheet` variable to know which sheet is currently active rather than resetting. In TypeCalc, the state variables are used to

- remember the scrolling position and evaluating cells within view (`scrollOffset`),
- storing and changing sheet names (`sheetNames`),

- switching between sheets (`activeSheet`), and
- to re-render the grid based on the window size after zooming in or out (`windowDimensions`).

Another similar React hook that the component uses is `useRef`, which also allows the front-end to remember values between re-renders, but without triggering a re-render when any of those values are changed (React, 2025f). The references managed by this hook are used by the `syncScroll()` function in order to synchronise scrolling between the grid body and its headers, as these are technically separate instances of `VariableSizeGrid`. The central React hook that enables most of `VirtualizedGrid`'s functionality is `useEffect`, allowing us to initialise, e.g., functions or variables when the component is initially mounted (React, 2025e). Our primary use for this is creating several event listeners that we have listening for various kinds of input required by functions, such as key presses or entering and leaving a cell.

To define the headers of the grid, we use the functions `ColumnHeader()` and `RowHeader()`, which are passed to `VariableSizeGrid` as its `itemData`. As parameters, they take a style, inherited from `style.css`, and either a column or row index. Each function then creates a `<div>` element with a label, which, for the rows, is the current index + 1, while the columns are given letters corresponding to the current index + 1, continuously adding further letters as needed, e.g., A-Z is 1-26, AA-AZ is 27-52, and so on. Though it would naturally have been easier to just use the index number for both axes, we had decided to use A1 notation in `TypeCalc`, meaning the columns are assigned letters instead of numbers. Our choice of A1 notation was mostly because of how widespread it has become among spreadsheet programs, starting all the way back with `VisiCalc` (Kapor, 1999). On top of that, we also appreciate A1 notation's inherent benefits, such as instantly making it clear which part of a cell address is the row and which is the column.

To actually define the visual grid itself, `VirtualizedGrid` returns various `<div>` elements for the grid body, headers, the `SheetFooter`, as well as the top-left corner between the headers (see Figure 43). Specifically, the outer `<div>` contains `SheetFooter` and two nested `<div>` with flexbox layout (Mozilla, 2025a). The first of these contains the header corner and column header, while the other nested `<div>` contains the row header and grid body. This division and decision to use flexbox layout made it easy to position the various components correctly because the column header and row header are `VariableSizeGrid` components of their own. By putting the corner and the top row header into their own separate flexbox, they are automatically arranged together above the rest of the grid. The `VirtualizedGrid` is placed between the `SheetHeader` and the `SheetFooter`, which are both positioned separately as defined in `style.css`.

```

1  return (
2    <div id="sheet">
3      <SheetHeader />
4      <div style={{ display: "flex" }}>
5        <div id="headerCorner" style={{ width: rowHeaderWidth - 1, height:
6          ↪ colHeaderHeight - 1, }}>
7          {""}
8        </div>
9        <Grid
10         columnCount={columnCount}
11         columnWidth={() => columnWidth}
12         height={colHeaderHeight}
13         rowCount={1}
14         rowHeight={() => colHeaderHeight}
15         width={width - rowHeaderWidth}
16         overscanColumnCount={5}
17         ref={colHeaderRef}
18       >
19         {ColumnHeader}
20       </Grid>
21     </div>
22     <div style={{ display: "flex" }}>
23       <Grid
24         columnCount={1}
25         columnWidth={() => rowHeaderWidth}
26         height={height - colHeaderHeight}
27         rowCount={rowCount}
28         rowHeight={() => rowHeight}
29         width={rowHeaderWidth}
30         overscanRowCount={5}
31         ref={rowHeaderRef}
32       >
33         {RowHeader}
34       </Grid>
35       <div id="gridBody">
36         <Grid
37           columnCount={columnCount}
38           columnWidth={() => columnWidth}
39           height={height - colHeaderHeight}
40           rowCount={rowCount}
41           rowHeight={() => rowHeight}
42           width={width - rowHeaderWidth}
43           onItemsRendered={() => EvalCellsInViewPort()}
44           overscanColumnCount={5}
45           overscanRowCount={5}
46           ref={bodyRef}
47           onScroll={syncScroll}
48         >
49           {GridCell}
50         </Grid>
51       </div>
52     </div>
53     <SheetFooter
54       sheetNames={sheetNames}
55       activeSheet={activeSheet}
56       setActiveSheet={setActiveSheet}
57       setSheetNames={setSheetNames}
58     />
59   </div>
60 )

```

Figure 43: VirtualizedGrid's return statement. Note that VariableSizeGrid instances are imported simply as Grid rather than the component's full name for brevity in the code.

10.2.2 SheetHeader Component

The `SheetHeader` is a stateless React component that defines the `<header>` element, which serves as TypeCalc's toolbar. It creates the various buttons, input fields, etc., of the header toolbar, along with the functions associated with these. The sole exception is `handleJump()`, which allows a user to 'jump' to a specified cell address. `handleJump()` is created as part of `VirtualizedGrid` because it makes scrolling to the designated cell work with the `VariableSizeGrid` component's own `scrollToItem()` function. The button and input field HTML elements used for `handleJump()` are still created as part of `SheetHeader`, however. The `useEffect` hook is used in `SheetHeader` similarly to how it was in `VirtualizedGrid`, creating the various event listeners needed for the component's functions. This primarily concerns any button clicks and input field interactions.

10.2.3 SheetFooter Component

`SheetFooter`, which is also a stateless React component, is responsible for the `<footer>` element along with any elements in it. Currently, the footer is exclusively used for the sheet selection feature, through which we can create new sheets and freely switch back and forth between any existing ones. It also comes with its own styling specifications to make it easier to distinguish the currently selected sheet from the rest. We have opted to keep the component stateless, as the props it is passed are all either state variables or those variables' setter functions, which are already always handled by `useState` in `VirtualizedGrid` itself. The state variables it receives are `sheetNames` to create a `<button>` for every sheet and allow the user to create new sheets, `activeSheet` to know and change which one is active, and `scrollOffset` to evaluate cells within the viewport when changing sheet. Any created sheets are returned as part of the `<footer>` along with a `<button>` element for creating a new sheet.

10.2.4 GridCell Component

The `GridCell` component defines the regular cells that make up the body of the grid by being passed to the `VariableSizeGrid` as its `itemData`, similarly to the header functions. The props that `VirtualizedGrid` passes to it are `columnIndex` and `rowIndex` to know where the cell is located, along with the prop `style` to inherit from `style.css`. The indices are also used to make up the cell's ID in A1 notation, by converting the column index to a number. `GridCell` is a stateful component, having two state variables and setters: `valueHolder` briefly stores the formula of a cell when entering it so that we can check if it has changed when leaving the cell, while `mySupports` contains an array of the cell's supporting cells that are currently within the viewport. Using these states along with various React event handler functions such as `onClick` and `onFocus` (React, 2025d), `GridCell` contains all the functionality required of the cells themselves. This, for example, includes updating the formula box's contents in the header and managing what happens when entering or leaving a cell. The event handlers are used to provide the functions enabling these features with required parameters, calling the functions when appropriate and removing any effects we do not want lingering. For example, when clicking on a cell, we may want to highlight which cells support it using `onFocus`, and then clear those highlights when entering a different cell using `onBlur`. Additionally, a lot of the features aiming to satisfy common use patterns that we went over in Section 5 are also enabled by `GridCell`, such as the various keyboard shortcuts,

or marking multiple cells. To create the visible, interactive cell in the grid, `Cell` returns a `<div>` element as an editable element with its HTML attributes `id` and `title` being equal to its `ID` field. It also comes with a single change in styling, as it colours the cell based on the row index. This gives the overall grid a striped look that helps differentiate between rows at a glance, as seen in Figure 44. Notably, this look is not present in either Google Sheets or Excel, and is a stylistic choice we made for TypeCalc. All the aforementioned React events are returned as part of the `<div>` element.

	A	B	C	D
1	10			
2	20			
3	=A1 + A2			
4				

Figure 44: Example of TypeCalc's striped look.

10.3 Interaction Schemes & Front-end Features

The widespread use of Excel and Google Sheets has set certain expectations for what is needed from a spreadsheet. This is not just in terms of looks, as is directly visible in our decision to use A1 notation, but also in regard to how the spreadsheet works and how it is navigated by the user. Given our unfamiliarity with front-end development, we once again looked to the two major spreadsheet applications to help identify the key features we should prioritise implementing in TypeCalc. Some of these we identified from the start, e.g., through our own prior experiences from working in spreadsheet applications, while others were discovered to be essential as we developed TypeCalc. The features we ended up choosing allow users to:

1. reset the workbook, or clear its contents,
2. jump to a specific cell (also allowing us to verify the dimensions of the grid),
3. navigate between cells with the arrow keys,
4. view the actual formula written in the cell and not just the result,
5. import an Excel 2003 XMLSS file and export workbook contents as either an XMLSS or a CSV file,
6. style text (make it bold, italic or underlined),
7. change colours of both text and the cell itself,
8. undo and redo cells,
9. see different sheets in a workbook as tabs that can be freely switched between,
10. show cell dependencies and dependents,
11. copy, cut, and paste cell contents, preserving the formula, and,
12. copy contents of one cell to a marked area of cells, and automatically adjust the formula

accordingly.

While some of these, such as the text styling, were a straightforward matter of creating a button and slightly manipulating pre-existing HTML functionality using TypeScript, others were more complex to implement. While some of these advanced features, e.g., exporting and importing, have been elaborated on in appropriate sections previously in the report, we wish to briefly highlight our solutions for points #9 showing cell dependencies and #11 selecting multiple cells and manipulating them.

10.3.1 Show Cell Supports and Dependents

One interesting feature that is well-known from Excel is the ability to see which cells directly support the currently selected cell. For example, if cell C4 contains the formula `=SUM(A1:C2) + B3`, then the cells in the cell area A1:C2 and the cell B3 should light up or otherwise have some indicator that they support B2 whenever B2 is in focus, as demonstrated in Figure 45. To accomplish this functionality in our front-end, we make use of the `ForEachReferred()` function from the back-end `Cell` class to retrieve all the cells that support the currently active cell. These cells are then highlighted by a CSS styling that adds a green outline. In this way, we deviate from both Excel and Google Sheets, which by default only highlight supported cells when writing in the formula box. Originally, this was done for testing purposes, but as we continued working, we found the constant highlighting a lot more helpful than it was intrusive, and so we decided to keep it. When another cell is put into focus, either via clicking another cell or using keyboard navigation, this border styling is cleared on every cell before the new supports are found. Some spreadsheets allow the user to see these connections in an even more explicit way by showing directional arrows between cells in the viewport. While we can find all the relevant connections and find the coordinates of these cells in React, we have not yet implemented this functionality in `TypeCalc`, and leave it for future work.

	A	B	C
1	1	2	3
2	4	5	6
3		7	
4			=SUM(A1:C2) + B3

Figure 45: `TypeCalc` highlights the supporting cells of the active cell.

We have also implemented the closely related but less famous feature of showing which cells depend on the active cell. We do this through the function `ForEachSupported()`, the sister function

of `ForEachReferred()` that is also written in the `Cell` class (Table 2). Creating `ForEachSupported()` was simpler, however, since much of the groundwork was already done. This is because we for each cell store cell reference ranges of the class `SupportRange` for all cells that depend on that cell. These ranges are placed in said cell's `supportSet` field, meaning that finding all cells that depend on a given cell is a simple matter of going through that cell's `supportSet`. On the front-end side, we opted to create the dependent highlighting as a feature that users can toggle on and off by pressing the CTRL key together with the "m" key. When the feature is toggled on, any cells that are supported by the active cell will have a dark blue, dotted line highlighting them (see Figure 46). This is unlike the highlighting of supports, which happens automatically. We decided to not automatically highlight supported cells, as it would otherwise be confusing to the user, since there is no intuitive way for new users to know what the differing highlight stylings indicate.

	A	B
1	123	
2	=B2	123
3		123

Figure 46: If toggled on, TypeCalc highlights the cells depending on the active cell.

10.3.2 Selecting Multiple Cells

One of the goals for TypeCalc was to allow interaction with one or more cells in the GUI and either copy, cut, paste, or delete the cells in the area. To end this, we wanted to solve two tasks, which in turn would make the back-end implementation for cut, paste, and delete easy to implement in the `Sheet` class. The first task was to dynamically highlight an area, showcasing the user the cells that they have currently marked, and the second task was to save this information in a smart manner, such that the area could be easily accessible. Our solution to the first problem was to use a nested for loop, which marked all the cells between the first cell that was selected when we began "highlighting" the cells and the current cell that was active. To ensure that we always get the correct area, we use the `normalizeArea()` method, which ensures that the upper-left and lower-right cells are found (see Figure 47). Then we used the nested for-loop to gather all the cell IDs - that is, their A1 notation, and retrieve their corresponding cells in the GUI. These are subsequently displayed with a light blue background. With the area marked, and the upper-left and lower-right cells being found in the prior step, we could simply store these. Here we decided to use `sessionStorage` - a JavaScript feature that allows JSON objects to be stored for the duration of a user session (Mozilla, 2025e). We decided to use `sessionStorage` because it provides an easy way to store information, which is most likely not going to be needed between reloads of the GUI

as opposed to `localStorage`, which persists between GUI resets (Mozilla, 2025d). Additionally, the use of `sessionStorage` is ideal for scenarios such as copying an area, since the need for manipulation of another area might not be immediate. In addition to this, having the `AreaCell` also made it possible to highlight all the cell IDs in the GUI. The function, which was used to perform this functionality, is called `setHighlight()`, and can be seen in Figure 47

```

1  function setHighlight(endCell: string, saveHighlight: boolean = false,
   ↪  arrowOptCol?: number, arrowOptRow?: number): void {
2      const endCellRef = new A1RefCellAddress(endCell);
3      let currentActiveCellRef = arrowOptCol && arrowOptRow ? new
   ↪  A1RefCellAddress(numberToLetters(arrowOptCol + 1) + (arrowOptRow +
   ↪  1)) : new A1RefCellAddress(ID);
4      let { ulCa, lrCa } =
   ↪  SuperCellAddress.normalizeArea(currentActiveCellRef, endCellRef);
5      clearVisualHighlight();
6      highlightSelectedCells(ulCa, lrCa);
7      if (saveHighlight) {
8          sessionStorage.setItem(
9              "selectionRange",
10             JSON.stringify({ ulCa: ulCa, lrCa: lrCa })),
11         );
12     }
13 }

```

Figure 47: The `setHighlight()` method, which is in charge of showcasing the selected area, and if chosen, copies it to `sessionStorage`

11 Testing

In the following section, we will provide a rundown of our chosen testing frameworks and why we chose them, as well as providing examples of how we used them.

11.1 Overview of Testing

An important aspect of software development is ensuring that the system works according to the specifications provided (Sommerville, 2016). While linting and syntax tools are good for avoiding syntactical errors in the code, they are incapable of detecting logical errors as a result of faulty code. Especially in an open-source project, where future users and contributors may continue on the TypeCalc project, it is vital to ensure that the codebase works as intended and methods are thoroughly tested, such that future contributors have a better ground for further development. This is where testing comes in. Testing is an invaluable tool for the process of demonstrating software functionality, and as a tool for detecting errors in the system (Uddin & Anand, 2019). Using testing, we are able to objectively determine if a piece of code works for a constructed scenario. Additionally, testing has been an essential part of our continuous integration practice, thereby strengthening our code quality as described in Section 4.3. Testing can be divided into three different levels: *Unit-testing*, *component-testing*, and *system-testing* (Sommerville, 2016). Unit testing is the lowest level of software testing; it tests a singular method or object in isolation to verify that it works according to its specification. Component testing is a more extensive type of testing that tests how multiple methods or classes work together in an isolated environment. System testing is the most involved way of testing a system. It regards comprehensive tests of components in a singular system. For unit- and component testing, we used a framework called *Vitest* (Vitest, 2025). The reasons why we chose Vitest as our testing framework are that it is simple to set up and use, and easily integrated into our application since it uses Vite. For our system testing, we used the Playwright framework, which allows us to simulate user interaction in the browser, such as inserting and evaluating formulas in the spreadsheet (Playwright, 2025). This type of testing is also called end-to-end, or E2E, and is meant to replicate an entire runtime of the program. In the following section, we will describe our implementation of unit-, component-, and system-testing.

11.2 Unit Testing

Unit-testing is about testing if a singular method or class of a program delivers the expected result, and was used in our project to ensure that classes and methods behaved as intended (Sommerville, 2016). When we started writing unit tests, our goal was to write at least one test for each of the methods and constructors in the back-end. This goal was not achieved, which we will elaborate on in Section 12. However, we did end up writing 168 unit and component tests, of which the majority are unit tests. We created the unit tests by having one test suite for each class using Vitest’s `describe()` method, and then, by applying a `beforeEach()` call, we configured a setup routine that automatically runs prior to each individual unit test within the `describe` block (Vitest, 2025). An example of a unit test is seen in Figure 48, where the test case has the name `Show Test` followed by a test function. In the test function, we expect the result returned by the `NumberCell.Show()` method to be “10” and to be of type `String`. The test passes as it should, and at an early stage of development, it acted as confirmation that this method worked as we intended. This proved useful throughout the entire project, as we after every change could test

that everything still worked.

```
...
test ("Show Test", () => {
  expect(typeof numberCell.Show(0, 0, fo)).toBe("string");
  expect(numberCell.Show(0, 0, fo)).toBe("10");
})
...
```

Figure 48: Extract from NumberCell.test.ts where a unit test of the NumberCell.Show() method is seen.

11.3 Component Testing

A software component is a larger group of units, and a component test verifies the interaction between multiple units (Sommerville, 2016). This is important because the units may function correctly when tested as individuals, but show unexpected behaviour once combined. Component tests serve as a middle ground between simple unit tests and comprehensive system testing. This became relevant for us when testing the functionality of ArrayFormula. To set up ArrayFormula.test.ts, we had to apply multiple methods from different classes for setting an input cell area, declaring a formula with a function call, recalculating the workbook, caching the formula and setting the output cells as array formulas. To better explain what we were testing for, see Figure 49. With this component test, we could assure that the back-end was functional in some aspects, and it was a stepping stone for more complex component tests involving the parser and end-to-end tests with the user interface.

A4	A	B	C	A4	A	B	C
1	1	2	3	1	1	2	3
2	4	5	6	2	4	5	6
3	7	8	9	3	7	8	9
4	=FREQUENCY(A1:C3, [2,4])			4	2		
5				5	2		
6				6	5		

Figure 49: FREQUENCY is a function in Formula.js that distributes values of an array or cell area (here A1:C3) into frequency bins. The bins are defined by an array of values which are [2, 4] in this case and will create the bins ≤ 2 , > 2 and ≤ 4 , and > 4 . The formula is shown to the left in cell A4. After evaluating the formula, the result is seen to the right in the cell area A4:A6.

11.4 System Testing

System tests provide a comprehensive test of the entire system, showcasing how the entire system works as intended in predefined scenarios. In our system testing, we have made use of Playwright to test if the entire system works for simulated test cases. While traditional testing measures provide a useful baseline for testing the validity of back-end functionality, they are not well-suited for checking how interactions work with the front-end GUI. In this case, it makes sense to use system testing, as it strives to simulate interaction in the same way that a user would. We chose Playwright as our tool of choice for system testing as it is easy to use, provides thorough documentation, and can be easily integrated into our CI setup. Our tests work by initially running the `vite dev` command, opening the server on localhost, and starting an instance of Chromium. From here, it tries to access the server on localhost, and interacts with the server as one could do manually with

the GUI. Playwright was useful as a means of checking if the GUI is reflecting commands correctly, and in particular, ensuring that custom commands, such as "CTRL + c", "CTRL + v", "CTRL + x" work as intended. To complement our system tests, Playwright provides a GUI of its own that shows how the test is executed in real time and what each command changes in the UI. To access this, we run the command `npx playwright test --ui`, and a GUI (as seen in Figure 50) will open.

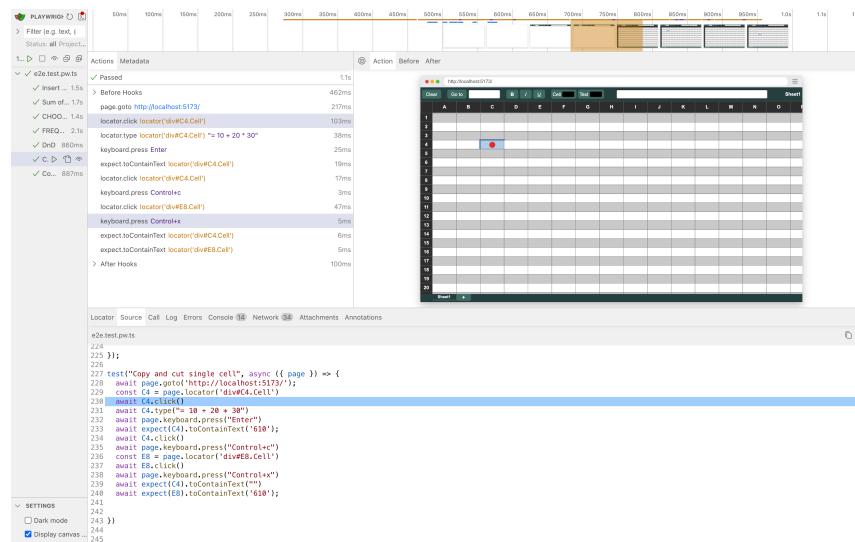


Figure 50: Playwright provides a GUI for debugging tests, and seeing how commands are performed in real life.

In the following section, we elaborate and reflect on our learning goals for testing, our testing coverage, and the value of our testing.

12 Reflections

Before concluding our report, we reflect upon certain aspects of the TypeCalc thesis project. We first address topics related to our implementation, these being the programming language differences, recalculation, cell reference expressions in formulas, undo/redo, our integration of React, shortcomings of react-window's `VariableSizeGrid` component, usage of the `Formula.js` library, and the approaches we have considered for extending and scaling up our API-layer. Afterwards, we reflect on topics related to the process of working on the project. This includes our use of Agile, our use of AI, the testing suite, TypeCalc's quality goals, and our intended learning outcomes. Lastly, we reflect on topics for future work on TypeCalc, providing an overview of notable bugs that we would prioritise fixing.

12.1 Reflections on the Product

Here, we reflect on TypeCalc as a piece of software, our choices in technologies, and the findings and considerations related to how the spreadsheet works.

12.1.1 Differences between TypeScript and C#

As mentioned numerous times, TypeCalc is based on the CoreCalc implementation by Peter Ses-toft. While building the TypeCalc implementation, we have made extensive use of elements from CoreCalc, often simply translating the code from C# to TypeScript. In general, these two coding languages share similar syntactical structure, such as class and method declarations. However, certain discrepancies are notable and interesting in terms of programming languages, which we address throughout this section. In Section 7.3, we describe how most of the inner logic and methods for classes, such as `NumberValue` and `TextValue`, could be simplified significantly. The reasoning behind this can be attributed to the type system that TypeScript uses and, by extension, the one that JavaScript uses, since TypeScript transpiles to JavaScript (Microsoft, 2025c). In JavaScript, we use a dynamic type system, which removes the need for explicit type declaration and type casting (Mozilla Developer Network, 2025a). By extension, this allows us to use more flexible types in TypeScript, since these are merely compile-time types that do not affect the runtime execution. In comparison to this, C# uses a static type system in which types must be explicitly declared at compile-time (Microsoft, 2025a). The benefit of this approach is that it helps in avoiding errors that might occur as a result of faulty type inference; which is one of the core ideals of the TypeScript superset. In that way, TypeScript is similar to C# while writing code, but varies during execution.

The fact that JavaScript uses a dynamic type system is also important for another aspect of TypeCalc. In TypeScript, we are allowed to use union types for variables, parameters and return values. Here we refer to declarations of these variables, parameters, or return values that allow them to exist as one of multiple types. This is again tied to the dynamic typing system, and for TypeScript - it allows more flexibility for the static type analysis that is performed. In Section 7.6, we briefly mention our decision to use specialised classes for cell references and cell addresses, albeit we do not provide the reasoning for this decision compared to the CoreCalc implementation. TypeScript forbids the use of constructor overloading. In C#, we can create multiple constructors for a single class, and each of them can be called in different contexts. For TypeScript, we are only allowed to

use a single constructor definition. To get around this issue, our first thought was to use inheritance and create these specialised classes that could allow us to work around the limitations on overloading. We saw the use of these specialisation classes as a way of keeping a cleaner structure for our references and addresses, since it would be easier to differentiate between the different notations being used. Later on, as we had gotten more comfortable in using TypeScript, we thought about other approaches to solving our initial problem. One solution to this was to use a combination of optional parameters and union types in order to simulate multiple constructors at the same time, which we did for our Sheet class constructor, as can be seen in Figure 51

```

constructor(workbook: Workbook, sheetName: string, arg3: boolean | number,
  ↪ arg4?: number, arg5?: boolean,
) {
    if (!arg4) {
        this.name = sheetName;
        this.workbook = workbook;
        this.functionSheet = arg3 as boolean;
    } else {
        this.name = sheetName;
        this.workbook = workbook;
        this.functionSheet = arg5 as boolean;
        this.cols = arg3 as number;
        this.rows = arg4;
        this.workbook.AddSheet(this);
    }
    this.cells = new SheetRep();
    /** rest omitted */
}

```

Figure 51: Sheet class constructor, which makes use of optional parameters and union types. The constructor needs to check for the existence of `arg4`, in order to determine which constructor to use, thus mimicking constructor overloading

While the use of these types could make our code significantly more condensed, since we do not have to create specialisation classes, we end up with code that is much less readable, and where we need to check for the existence of certain variables to enter the correct constructor option. Lastly, since part of our goal with TypeCalc was to create a GUI, the fact that TypeScript transpiles to JavaScript was a great benefit, as JavaScript is highly compatible with browsers, being by far the most common programming language for browser applications (W3Techs, 2025).

12.1.2 Cell reference expressions in formulas

We want to highlight an important distinction between how CoreCalc and TypeCalc handle reference expressions. CoreCalc has a memory optimisation in which it allows multiple formulas in separate cells to point to the same cell reference expression in the representation. This essentially means that a large amount of formulas can share the same reference expression, which can potentially save a very large amount of space, for example, when formulas are copied to large cell areas. Conversely, TypeCalc saves a new reference expression every time a formula cell is copied, as well as the string of the user input to the cell whenever a user changes the formula of a cell. This choice was made in part because of an initial misunderstanding of how CoreCalc functioned, and ultimately deemed too extensive to fix at the late stage in development at which we properly understood our mistake.

While we would prefer to use the CoreCalc optimisation in a later release of TypeCalc, our solution is not solely disadvantageous. The fact that we save formula strings means we can omit the use of the somewhat complicated `Show()` function in CoreCalc (Sestoft, 2014b, p. 41) and instead save a direct copy of the formula to be shown in a cell when it is hovered. We also argue that our solution is more instinctive in that it more directly reflects the meaning of the word "copy" - i.e. the creation of a duplicate (Merriam-Webster, 2025). While this point is not necessarily attributed much value, it does have some merit in an open-source project that is meant to be picked up and iterated on by others.

The consequences are, however, meaningful. Our simpler representation means that the copying of formulas is done via matching regex expressions and changing them to account for the new offsets in the spreadsheet A1 notation, such that the formula `=B1` is changed to `=B2` if the formula is copied to the cell below the original, and `=C1` if it is copied to the cell to the right of it. This is not only inelegant, it also induces bugs. For example, we currently have a bug with the copying of absolute R1C1 references, where a reference expression in a formula, such as the one in `=R1C1` will be transformed to `=R2C2` if the cell is copied to the row below. While not exactly a difficult bug to fix, this highlights some of the unwanted behaviour that arises from our decision. Additionally, we lose the previously mentioned memory optimisation, which can be very costly, especially considering how essential the copying of formulas is in spreadsheet usage. In retrospect, we would definitely use the CoreCalc optimisation if we were to recreate TypeCalc today, or possibly in a future release of the software.

12.1.3 Recalculation

A problem occurred in TypeCalc's recalculation when importing a large XMLSS file with 10,000 chained cell references in it, resulting in a `RangeError: Maximum call stack size exceeded`. The 10,000 cells created a chain of dependencies where A1 evaluated to a constant 10, and A2 would then reference cell A1 with the formula `=A1`, and this pattern was repeated down to cell A10000, which referenced cell A9999 with the formula `=A9999`. The `RangeError` occurs because we recursively mark Dirty the support sets of each edited cell, and when there are 10,000 edited cells, the call stack exceeds its maximum size, i.e. the recursion becomes too deep (MDN, 2025). Therefore, we tried following SIT's suggestion for applying full recalculation when importing files (Sestoft, 2014b). At first, this solution worked, and the problem seemed to be solved by calling full recalculation when a `RangeError` occurred. But then we started wondering why this solution worked and why the call stack did not exceed in a full recalculation, which also initiates recursion by evaluating all active cells, which then have to evaluate their expressions. Therefore, we tried reversing the chain of dependencies in the imported file with 10,000 cells such that the chain started in cell A10000 and ended in cell A1, and this again resulted in a `RangeError`. To sum up, we discovered that full recalculation can recalculate a chain of 10,000 cell references starting in A1 and ending at A10000, but not when starting in A10000 and ending in A1. Thus, full recalculation did not resolve our problem fully, and this may occur because of the order in which active cells are stored and iterated through. Nevertheless, this problem will be added to the list of notable bugs, as it is a problem that should be looked into (see Section 12.3.1).

12.1.4 Undo/Redo Functionality

As described in Section 7.7, we implemented undo/redo functionality in TypeCalc, which reverts (CTRL+z) or re-inserts (CTRL+y) one cell from the sheet at a time. This functionality works when single cells are edited, evaluated or copy-pasted, but a problem occurs when formulas evaluate to whole cell areas or if a cell area is copy-pasted. If a user, for example, copy-pastes a cell area and wants to undo the action, then the cells of the pasted area are undone one at a time, rather than all being undone instantly. This might conflict with the user's anticipation if they expect that pasting a cell area of e.g. nine cells can be undone with one single click on CTRL+z (Nielsen, 1994). In TypeCalc, it would currently take nine clicks on CTRL+z to remove a cell area of nine cells. This is because `history` array for managing undo/redo caches each cell insertion individually, and undo/redo is only called once per click on CTRL+z or CTRL+y. We therefore add this problem to the list of notable bugs (Section 12.3.1) as it is a drawback in the user experience of TypeCalc.

In retrospect, we thought of solutions for the undo/redo bug, which can be implemented by future contributors. One idea is to add a field to the Sheet class called `cellsPerAction` of type `number[]`. The purpose of `cellsPerAction` would be to log how many cells were changed by each action made by the user. So if the user, for example, pastes a cell area of nine cells into the sheet as their first action, 9 will be logged at index 0 of `cellsPerAction`. Then, when the user presses CTRL+z/y, we should check the value at index `historyPointer` of `cellsPerAction`, and then call `undo` or `redo` the amount of times stored in `cellsPerAction` - e.g. nine times if the value is 9. In this way, undo/redo would appear to follow the number of actions taken by the user in the GUI, rather than the number of edited cells in the back-end, and we believe this would result in a better user experience.

Late in the project, we discovered a particularly critical flaw in our undo/redo implementation. The problem occurs when undoing cells with dependents, at which point the dependents are not properly recalculated. For example, if the formula `=A1` is written in cell B1 and 10 is then written in cell A1, then cell B1 will correctly evaluate to 10. But if CTRL+z is then pressed by the user, cell A1 will be blank, and cell B1 will still, now falsely, show the value 10. The problem stems from the fact that we directly set `BlankCells` via `SheetRep.Set()` rather than via `Sheet.SetCell()`, thereby failing to properly update the support sets of undone cells. This problem will be added to the top of the list of notable bugs (Section 12.3.1) since the undo/redo feature is such a common feature that it would most likely be a bug that users encounter frequently. We are aware that this bug violates the correctness and consistency of TypeCalc's recalculation implementation, which is another reason for adding it to the very top of the bug list.

12.1.5 React Integration

As alluded to in Section 10, it can be said that we stumbled into working with the React library as the foundation of our GUI due to finding `react-window`. Fortunately for us, this proved to be a blessing in disguise. React allowed us to use the `react-window` library itself, whose virtualisation would prove key in enabling our GUI. On top of that, its own features would help us a lot, given our lack of experience with front-end development. Key to this are React's event handler functions. Not only is there a great variety of these, allowing us to cover many use cases for when and how to trigger specific interactions, but most event handler functions also have easily understandable names; there was no confusion about when, e.g., `onClick()` would be useful. Using these, we were

able to quickly create features like key-bindings, displaying the current cell's ID, or highlighting support cells in the GUI.

Another benefit of React is that it is structured around components such as the `VariableSizeGrid` component from `react-window`. This modularity meant we could safely create new components without breaking the rest of the GUI, allowing us to experiment and learn through iteration and trial and error. In a similar vein, allowing us to divide the front-end into components also made it more approachable for people to start working on the front-end. You could start by learning the relevant component at first and then slowly learn how it all fits together as you worked on different features, which creates a more satisfying learning curve and also simplifies the division of labour in a group. Additionally, having child-components for `VirtualizedGrid` allowed us to keep the code of each component a lot more focused and lean, as component files only ever had to contain the functions or methods that they directly need, with a few exceptions like the briefly mentioned `handleJump()` function.

One consequence of learning while doing is that our front-end did not utilise React that well initially, and so we have had to periodically go back and refactor old code to apply React's features. This included work such as breaking a class down into components or rewriting classes to instead be components and later establishing appropriate parent-child relations between said components in order to pass on props. This cost us considerable time and rarely provided any immediate gains in performance or functionality. With that said, it is not as if we chose React over another library for building user interfaces - as mentioned, none of us had any prior experience with front-end development, and so we would likely face similar issues regardless of approach. Ultimately, we believe that the upsides of React outweighed any setbacks, and we would use React again in future front-end projects.

12.1.6 Shortcomings of `react-window`'s Grid

Using `react-window`, we were able to very quickly scale up the size of `TypeCalc` all the way to, and exceeding, the desired 64,000 columns and 1,000,000 rows. That said, we encountered two notable issues that had to be accounted for and worked around. These issues were present regardless of which of `react-window`'s two grid components we used. Firstly, spreadsheet applications generally allow you to always see the column headers when scrolling vertically, and the row headers when scrolling horizontally, never letting them scroll out of view. To emulate this behaviour, we wanted to make those headers 'sticky', as CSS calls it. Using a single grid, this is not supported in `react-window`, as we cannot make a singular row and column sticky. Instead, we created a workaround where these two headers are created as individual instances of `VariableSizeGrid`, along with a separate cell that functions as the corner between the two. The rest of the grid is then rendered separately from these. This created scrolling-related issues, however, as each grid element has its own scrollbars. We were not able to make individual scrollbars hidden, so we deemed the less intrusive option would be to hide them all rather than showing them all, since users can still orient themselves using the headers and scroll via peripherals. Additionally, the separation into individual grids meant we had to synchronise scrolling between the headers and the corresponding axis of the main grid. This works fine if only the headers have to synchronise scrolling with the main grid, or vice versa, but enabling both at the same time will continuously reset the scroll position as soon as you begin scrolling the other axis. Again, we opted for the lesser of two evils, so currently, scrolling is

only possible on the grid body, but the headers will synchronise. It should be noted that there is an example implementation for a react-window `List` component that successfully combines headers and body into a single list while preserving the sticky behaviour (Vaughn, 2025a). We tried to translate this into TypeCalc’s implementation of the `VariableSizeGrid` component, as it could potentially solve the problem with scrolling since we would no longer have separate grids that had to synchronise with each other. Due to time restrictions for our project, however, we decided to put it on hold and prioritise tasks more crucial to GUI functionality, leaving the resolution as a task for future work.

The second notable problem is that the number of pixels you can render is limited by your browser. This means that regardless of the system, the amount of pixels that can be rendered is finite, which means that rows can only have a certain height, or there will not be space for a million of them in the browser. Currently, we have defined rows to be 30 pixels tall, which limits us to having 894,784 rows rather than a million. We verified that this is indeed an issue by temporarily making the rows only 20 pixels tall, which allowed us to easily fit in the million rows. Another solution we considered was to only reduce the height of rows that are outside of the viewport, allowing us to fit more rows inside the pixel maximum. The downside of such an approach is that we would need to redraw cells to the full size whenever they come into view, hindering performance. While this would probably still be an effective short-term solution despite the resizing, it may lead to complications in future development as a feature like resizing of rows and columns is implemented. We briefly considered potential ways to work with this future feature in mind, e.g. by enforcing a maximum row height and storing any resizing values so that it may be reapplied once the row comes into view again. Ultimately, we decided to let it be for now as we were aware of the issue and are still able to produce up to almost 900,000 rows in the GUI with our current cell height. Practically speaking, the back-end sheet representation still supports the full sheet size, which means you can refer to a value that cannot be shown in the GUI unless the row height is changed.

12.1.7 On the Integration of Formula.js

In Section 8.1, we described our usage of Formula.js for implementing spreadsheet functions in TypeCalc. The reasons for doing so were to avoid code translations from CoreCalc and because Formula.js contains a larger amount of implemented functions. Formula.js is also written in JavaScript, making it directly compatible with TypeCalc. While it did spare us from translating roughly 1200 lines of code from the `Function.cs` file from CoreCalc, we ended up extending TypeCalc’s `Expressions.ts` file by roughly 400 lines compared to CoreCalc. Naturally, the newly added 400 lines probably demanded more consideration of us as developers, which makes the comparison in workloads between this approach and the translation of CoreCalc’s `Function.cs` file blurry. Consequentially, the workload cannot be measured by the amount of written code, but rather by the time that goes into writing the code, which may very well have been larger. In return, we gained access to a large, well-tested, and updated library of 399 spreadsheet functions (Ghalimi et al., 2025a), which we believe adds to the quality of TypeCalc.

In terms of the compatibility between TypeCalc and Formula.js, we encountered no issues as TypeScript transpiles to JavaScript and Formula.js is written in JavaScript. The challenge was to convert the object types of TypeCalc, such as `NumberValue`, `BooleanValue`, and `ExprArray`, to primitive types such as `number`, `boolean` type `true` or `false` and array types such as `number[]`. However,

the functions of CoreCalc also had to maintain such conversions from object types to native types of C# when passing arguments to functions, so in this sense, TypeCalc does not do extra work compared to CoreCalc. The difference is that TypeCalc has to do it before passing the arguments to Formula.js, whereas CoreCalc does it afterwards inside the function declarations. We did not immediately recognise any problems with this difference in the software architecture, but our view on the subject changed once Formula.js was updated. After we had stopped developing TypeCalc, the contributors of Formula.js released a new version, resulting in more functions being added and a more thorough documentation on their home page (Ghalimi et al., 2025a), which highlights the problems related to this discrepancy. If new non-strict functions are added to Formula.js, then it will be problematic that we evaluate all expressions before we pass them to the formula.js functions, as this will violate the purpose of non-strictness. Therefore, we should look into which functions of Formula.js are non-strict and then implement them directly in TypeCalc such that we respect their non-strict nature, just like we did for the functions `IF(...)` and `CHOOSE(...)`.

The update of Formula.js also meant that functions such as `TRANSPOSE(...)`, which transposes the columns into rows and rows into columns of a cell area, were also added to TypeCalc, which we had not accommodated for. As mentioned in Section 8.6, only `FREQUENCY(...)` and `MODE-MULT(...)` return array-valued results in TypeCalc, but that changed when `TRANSPOSE(...)` was added to Formula.js. `TRANSPOSE(...)` only works partially in TypeCalc because the `Fun-Call.Eval()` is currently caching array-valued results in a column-based manner (as described in Section 8.6), so `TRANSPOSE(...)` can only transpose a single horizontal row of values to a vertical column and not the other way around. Therefore, TypeCalc's evaluation of function calls needs some structural changes to accommodate new functions such as `TRANSPOSE(...)`, and this will be added to the list of bugs (Section 12.3.1).

One way of avoiding the problem with new releases of Formula.js could have been to keep the old version of Formula.js in TypeCalc, which we considered, but decided not to do for several reasons. First of all, the contributors also fixed bugs and updated the documentation of the functions, which were positive updates for TypeCalc (Ghalimi et al., 2025b). Secondly, we wanted future implementations of already compatible functions to be implemented in TypeCalc, such that if a simple function were added to Formula.js which takes a number as argument and returns a number, this should immediately work in TypeCalc. Therefore, we decided to dynamically update the imports of Formula.js rather than sticking with an older version that would not be updated. This decision, however, has its pros and cons. On one hand, it leaves us with at least one more bug because `TRANSPOSE(...)` is imported but does not work properly in TypeCalc. On the other hand, TypeCalc will benefit from the updates of Formula.js that does not require refactoring of TypeCalc's source code. We believe that our decision is suitable for an open-source project like TypeCalc because its purpose is to let future interested developers contribute to the project. Together with a list of notable bugs, this provides a better and more honest starting point.

12.1.8 Extension and Scaling of API-Layer

In anticipation of later and more extensive use of the TypeCalc back-end, we consider two approaches to scaling up our API-Layer depending on use cases. While we have not implemented these considerations in this version of TypeCalc, it serves as a priority for future releases of TypeCalc. Our initial consideration was to add the TypeCalc API-Layer to common package hosting

sites, such as the nodejs package manager (Github, 2025b) which would allow users to work with TypeCalc in their own spreadsheet implementations, albeit it would require some restructuring of our project folders and methods to be fully functional and valuable for those using our API. The downside of this is that the user needs to handle all processes locally, however, they would be able to more easily add additional features or propose changes to the open-source implementation. Our other consideration for scaling was to introduce a server endpoint, which could process HTTP requests relating to different operations, such as creating a new workbook, filling in a cell with a formula, etc. Compared to our first solution, this would potentially alleviate the user from having to spend their own processing power on handling requests, instead relying on the server's ability to process tasks and save user data, such as cell content, server-side.

12.2 Reflections on the Process

Here we reflect on our experiences, decisions and learnings related to the process of writing TypeCalc and this thesis.

12.2.1 On the Use of Agile

In Section 4.1, we described our use of Agile principles and how we used a simplified version of a Kanban board. In this section, we reflect upon the idea of Agile as a software development planning approach and in extension to this, the value of the Kanban board in our project. We chose to use an Agile development approach since it has become standard within the field of software development, and the approach to software development that we had been introduced to through our curriculum. One of the core ideals of Agile is being able to adapt to different needs and team constellations, such that the team can work effectively instead of following a rigorous planning framework. For our project, we decided to use the Kanban planning approach, since it is relatively forgiving for inexperienced users, as its primary purpose is to provide an overview of the tasks that needs to be done and by whom, and it can be easily integrated into already existing team practices. Furthermore, given the length of this project, and the amount of work needed, it made sense to use a framework such as Kanban (Asana, 2025).

Using the Kanban board, we could get a better overview for the tasks we needed to complete, and what the next sets of task were going to be. We did however, encounter some problem with task specifications, where most tasks were discussed on group meetings, and as a result of this, we did not dedicate as much time to create task specifications on the Kanban board. The reasoning behind this could both be attributed to our inexperience with using Kanban, but definitely also the fact that our group spent much time in meetings, and therefore could address the task specifications verbally. The structure of our project, which initially required a lot of time on translating CoreCalc specifications into TypeCalc, made task specifications somewhat simple at first, as tasks were often along the lines of *Rewrite X from CoreCalc*. As such our early impressions of Kanban were mostly that it served as a tool for overview of which tasks currently existed. As our project progressed on, we unfortunately did not get into the habit of using more in-depth task specifications.

In the methodology section, we shortly address the fact that our Kanban board only used three columns, compared to the standard five columns. The two missing columns in our Kanban board was the ready for review column and the backlog column. One of the core ideas of Kanban is that the entire group can get an overview of what needs to be done. In our project, we initially tried to

break the project down into smaller bits, which ended up being the foundation for our quality goals. However, as the project moved more in the direction of building on CoreCalc, we found ourselves at a crossroad, where the tasks we worked on were not directly equivalent to the actual project goals. Likewise, the implementation of continuous integration was also partially responsible for the removal of the ready for review column, since the introduction of automated tests ended up being our way of ensuring that the code was up to our quality. In hindsight, review is important and we may have placed too much trust in code made by a single group member and the limited tests that were run as part of our CI.

On a more positive note, the introduction of Agile methods such as pair programming and CI has been a highly positive experience for our project. Pair programming was essential in overcoming several problems, and in turn provided a better understanding of the problem for both individuals. Additionally, the use of continuous integration provided a better ground for decision making, and the ability to feel more confident when making a pull request. One of the key notions of continuous integration, and by extension DevOps, is fostering a culture where people feel confident in exploring and adding new features to a system without the backlash of pushing broken code to the codebase (Gift et al., 2019). For us the introduction of automated tests were a key part in doing this.

If we were to start this project again, we would have done several things differently related to the use of Kanban. For one, we would have spent more time on assessing the requirements of the project, breaking all of these down into actionable tasks, and ensuring that clear task specifications were added. Furthermore, we would have dedicated more time for doing code reviews, and not just relying on the CI tests, which only reveals what we tested for. This, in turn, would also ensure that we had a much stronger foundation for subsequent code additions. For future research and work that includes the implementation of larger software projects, we want to emphasise the need for proper project management tools. We did not use our Kanban board optimally, which led to frustrations and confusion at times. Coming from primarily smaller projects in smaller groups during our master’s programme, the importance of structure and management for these more extensive projects has become evident.

12.2.2 On the Value of Generative AI

In Section 4.4, we describe our use of AI tools throughout the TypeCalc project. In this section, we want to reflect upon the implications of using AI tools for learning a new language and the increasing relevance of these tools in professional environments. Coming into the TypeCalc project, none of us had any real experience working with TypeScript. While TypeScript as a language shares many similarities to other programming languages we have worked with, such as Python and Java, there has still been quite a steep learning curve in adopting TypeScript. Here we found the use of AI tools useful, since it allowed us understand differences between TypeScript and other languages faster, and how to overcome these differences. Being able to obtain assistance in this manner was a major contributor to our understanding, and improved our rate of learning the TypeScript syntax.

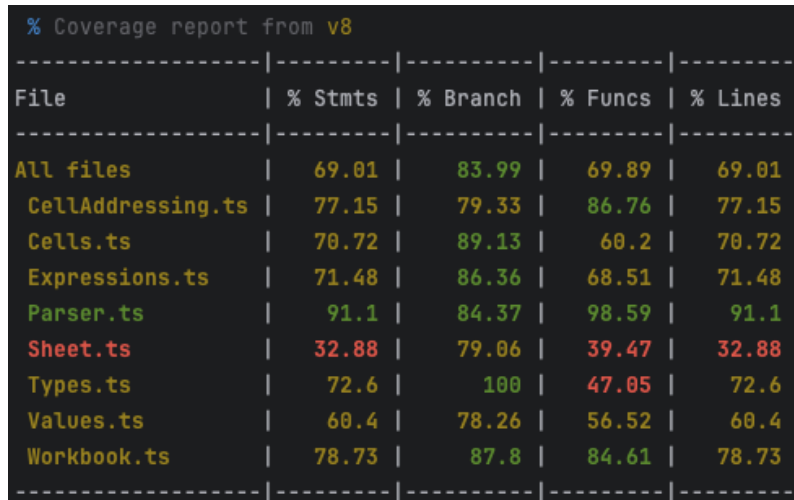
Furthermore, while AI tool solution outputs should be taken with a grain of salt, we argue that being able to recognise faulty solutions, and learning how to navigate between hallucinated outputs, also become a learning opportunity. Since the conception of AI tools, such as ChatGPT, in late 2022, the workflows of many industries have seen significant changes as a result (Kimbrough, 2025). In particular, the field of IT has seen a massive disruption as a result of these tools. Becoming

well-acquainted with these tools, and being able to recognise their strengths and weaknesses seem essential for becoming part of the next generation of AI-enabled IT professionals.

12.2.3 Testing

TypeCalc has a total of 168 unit and component tests in Vitest and 10 system tests in Playwright. The tests have aided our development process by confirming that the code we write operates as expected, and when new functionalities are added to TypeCalc, we use the tests to confirm that the old functionality is still intact and compatible with the new features. However, we ended up writing fewer tests than we had initially set out for. We wanted to write at least one unit test for each constructor and method of TypeCalc, and ideally, we also believe that there should be more component and system tests to better verify the expected behaviour. A lot of the unit tests are used for testing the behaviour of different Formula.js functions with TypeCalc, and therefore do not test different methods and constructors.

One of the key metrics used when it comes to testing of software is test coverage. Test coverage helps software developers understand how well their tests cover the methods, functions, and lines that exist within the codebase. For TypeCalc, we have used the terminal to generate a coverage report through our testing framework Vitest, which can be seen in Figure 52.



File	% Stmts	% Branch	% Funcs	% Lines
All files	69.01	83.99	69.89	69.01
CellAddressing.ts	77.15	79.33	86.76	77.15
Cells.ts	70.72	89.13	60.2	70.72
Expressions.ts	71.48	86.36	68.51	71.48
Parser.ts	91.1	84.37	98.59	91.1
Sheet.ts	32.88	79.06	39.47	32.88
Types.ts	72.6	100	47.05	72.6
Values.ts	60.4	78.26	56.52	60.4
Workbook.ts	78.73	87.8	84.61	78.73

Figure 52: Vitest Coverage Report of back-end files in TypeCalc.

Our Vitest coverage report reveals that we cover 69,89% of all the methods and constructors in the back-end of TypeCalc with Vitest. Google’s testing team puts this in the range of “acceptable” code coverage, but less than what is “commendable” or “exemplary” (Arguelles et al., 2020). While our coverage is lower than what we aimed for, we argue that our lower score can be attributed to two things:

- TypeCalc is based on the CoreCalc implementation described in (Sestoft, 2014a), containing several methods used for debugging and getter/setter methods. Many of these are less important to test.
- Furthermore, some of the implementation left in TypeCalc pertains to future enhancements

that we did not manage to implement for TypeCalc version 0, such as being able to insert and remove columns and/or rows. These are included in the code for future work, but are not necessarily tested.

We believe this reflects a decent level of coverage but recognise that we should strive for more to achieve our own goals of testing the software. Thus, a goal for future work on TypeCalc is to achieve 75% to 90% test coverage for constructors and methods (Arguelles et al., 2020).

According to Sommerville (2016), generic software products, which are products that are not developed for a specific company or user segment, should have tests for all features of the system. From this definition, we categorise TypeCalc as a generic software product. TypeCalc currently does not live up to this standard of feature testing, as there are, for example, no system tests for undo/redo. Again, this confirms that the coverage of our tests should have been higher; all features of TypeCalc should have at least one system test. Overall, these reflections indicate that we did not fully achieve our eighth quality goal, implementing a testing suite for demonstrating the functionality of TypeCalc. However, it is important to acknowledge that the 69,89% of coverage (see Figure 52) helped us catch several bugs before pushing new functionality and that it adds a reasonable level of software tests to TypeCalc.

In hindsight, we believe that Test-Driven Development (TDD) may have been a suitable approach for this project (Sommerville, 2016). TDD is an incremental development approach where testing and code development are interleaved such that each piece of code must pass its test before developers can proceed to the next task (Sommerville, 2016). This approach would have required us to write tests for all the functionalities of TypeCalc and thereby result in much higher test coverage. The process of TDD is special because it involves writing each test before the tested functionality has been implemented. The argument for doing so, is that by starting out by writing the test that should verify a functionality, developers will reach a better understanding of what that certain functionality should do (Sommerville, 2016). This approach is the opposite of what we did when developing TypeCalc, where we would first implement a functionality and then afterwards test it. Maybe, if we had followed TDD, we could have achieved a better understanding of the functionalities we wished to implement, which maybe could have lead to fewer bugs during development. Especially in projects that include a large amount of code translation, such as what we did from CoreCalc to TypeCalc, we believe that tests can be proactively made to better determine the correctness of the implemented code. Putting a higher emphasis on testing than we were able to throughout this project should be a priority.

12.2.4 Retrospective on Quality Goals

We achieved 15 of our initial goals, though not all of the first 13, with goals #6 and #8 being partially completed. Goals #6, implementing undo/redo, and #8, implementing a testing suite, have already been reflected on extensively, though we wish to touch on our shortcomings in regards to goal #8 in relation to our handling of our quality goals. These failings ultimately stem from us not prioritising the testing suite as much as we should have as the project progressed and other parts were considered more pressing. Given that we completed the 13th goal of implementing algorithms and data structures to handle larger spreadsheets ahead of time, it would have been appropriate to dedicate time to any prior but still unfinished quality goals. In this manner, the testing suite could

have been given the extra resources it currently lacks. For goal #17, implementing autofill in the spreadsheet, we wish to clarify the wording, as it was potentially unclear. Autofill was understood at the time it was written to mean that formulas containing one or more relative cell references would automatically adjust when pasted to multiple cells at once, rather than just copying as though it was an absolute reference. Upon looking into this, we found that by 'autofill', Excel means the ability to "fill cells with data that follows a pattern or are based on data in other cells" (Microsoft, 2025b, para. 1). For example, if we have two cells, A1 and A2, that contain the values 1 and 2 in Excel, then by selecting those cells and dragging the so-called fill handle down 2 cells further, Excel will recognise the pattern and fill out the cells A3 and A4 with the values 3 and 4, respectively. This is a notable diversion from our understanding of autofill in goal #17, and so is a breach of TypeCalc's external consistency (Nielsen, 1994). In terms of the goal we intended to set for ourselves, we consider this accomplished.

Part of the purpose of the quality goals was for us to show awareness of our own limits and what we could realistically accomplish within the scope of a thesis project, but also to reflect on what further development could look like. This is particularly relevant, given that TypeCalc is meant to be an open-source project that others can continue building on. With this in mind, the fact that we managed to fulfil more goals than expected speaks to us being too short-sighted or having a bias towards listing goals we thought of as realistically achievable. This was further exacerbated by our misjudgment of several goals' difficulty. For example, increasing the grid size from 20 x 20 all the way to 16,000 x 1,000,000, and then even 64,000 x 1,000,000, was trivial thanks to the libraries we ended up using for the front-end implementation. In conjunction with the QT4, this makes goal #7 obsolete. Another example was goal #12, which was achieved simultaneously with goal #4 just by us using HTML, CSS and TypeScript. On the other hand, a function like undo/redos was a lot more complicated than its sixth position on the list would indicate.

Another important variable that we did not properly factor in when writing the initial list of goals was the head start we would get from having access to the CoreCalc source code. This meant that we were not actually building the back-end entirely from scratch. We could instead translate a lot of the CoreCalc code from C# to TypeScript. As a result, we largely ignored the ordering of the quality goals, instead prioritising the translation of crucial parts of CoreCalc and any dependencies. That said, utilising CoreCalc in this manner was by no means a mistake, and though the process of translating code is not the same as writing it from scratch, it still allowed us to gain a thorough understanding of all the moving parts needed to implement a spreadsheet and how said parts fit together. It was also a process where we had to make decisions about, e.g., which parts of CoreCalc to not include, or how to best translate C# code to TypeScript. What we should have done instead was revise the quality goals to fit this change in approach. If we were to start over now, a more appropriate list of quality goals could look like this:

1. Develop a simple prototype of a spreadsheet with a grid size of 20x20.
2. Implement basic functions such as ADD, SUM and MINUS.
3. Implement the prototype in a range of different web browsers.
4. Implement cell formatting features (e.g., text styling, colour styling).
5. Implement a suite for testing functionality and performance that is updated continuously.
6. Increase the grid size to 64,000 x 1,000,000 or more and implement algorithms and data structures to handle the increased size

7. Allow users to import and export a common spreadsheet file format, such as XML.
8. Allow the sheet to show cell dependencies and cell supports.
9. Implement advanced functions such as FREQUENCY or MODEMULT.
10. Allow for automatic recalculations of cell operations
11. Allow for multiple sheets that can reference each other.
12. Implement undo/redo functionality for spreadsheet operations.
13. Allow relative cell references to be automatically adjusted when pasted to a different cell.
14. Make the cells draggable.
15. Implement and thoroughly test every function from Formula.js
16. Allow for resizing individual rows and columns.
17. Implement visual diagrams such as single-line diagrams and histograms.
18. Allow for importing and exporting of additional file formats.
19. Implement autofill to fill cells with data, following a pattern from data in other cells.
20. Allow users to collaborate in the spreadsheet.

Having condensed our completed and partially completed goals down to the first 13 goals, this revised list is more closely aligned with our stated minimum. Though we did not adequately satisfy our goals for the testing suite, we found it appropriate to place it as an early goal in the new list, having moved it from goal #8 to #5. This is because we believe it would be beneficial to start work on a testing suite as early as possible to follow the TDD approach we touched on in Section 12.2.3. It also rearranges the three goals we did not accomplish at all together with four new ones in what we believe is a more realistic path of progression for future development.

12.2.5 Learning Outcomes

Lastly, we will briefly reflect on our intended learning outcomes, which were:

1. Utilise HTML, basic CSS, and TypeScript, as well as UI principles to develop a dynamic application.
2. Describe the processes and challenges involved in implementing spreadsheet functionality.
3. Implement efficient data structures to manage spreadsheet data and optimise performance for large datasets.
4. Apply algorithms to handle formula evaluation, dependency tracking, and recalculation of cells efficiently.
5. Thoroughly document implemented functionality and design choices for an open-source project on GitHub.
6. Perform extensive testing to evaluate and document code quality by creating a test suite.
7. Organise and manage a medium-sized software development project by applying Agile principles and iterative development methodologies.

As one can surmise, most of our goals were closely intertwined with the process of working on TypeCalc. For example, to create the GUI and satisfy learning outcome #1 we would use HTML, CSS and TypeScript together with UI principles to help shape how we wanted it to work and

function, but to have a responsive GUI we had to be able to fetch data from the back-end almost instantly. To that end, we needed efficient data structures with performance optimised for large data sets and algorithms that could handle crucial functionality like formula evaluation, recalculation, and dependency tracking, which our cells depend on. As a consequence, we also made progress in regards to the 3rd and 4th outcome. Additionally, because of the project's scale, we wound up utilising methods for facilitating software development in a group that we had learned in prior semesters, thereby working towards outcome #7. Because all of this was done as part of a master's thesis project, learning outcome #5 of providing our Github project with thorough documentation of functionality and design choices is also due to happen by the time this report is finished, as it will be uploaded to our repository along with the code documentation made with JSDoc. Furthermore, this very report is our primary contribution towards fulfilling outcome #2, as we throughout the report describe the processes and challenges we experienced during the process of creating TypeCalc. Among our stated learning outcomes, the greatest shortcoming is outcome #6 - our testing suite, as has already been discussed throughout this section. That said, the point of the learning outcomes is, after all, to learn. From this stumble we have learned both the importance and the work required to maintain an extensive testing suite, and so can take a better approach in the future. This speaks to us ultimately living up to outcome #6 as a learning goal.

12.3 Future Work

As has been mentioned throughout the report, we have had to prioritise certain features in favour of others, given that TypeCalc was created as part of a master's thesis project. However, there was no clear cut-off point for which feature we would stop after, so we all kept working until our self-imposed deadline. We made an effort between that deadline and the hand-in date to fix up the most pressing issues, but ultimately ended up with loose ends. This was both in terms of some features still being bugged, whilst other features were planned by us but never actually started. In this section, we present an overview of notable bugs and issues that we have identified and should be prioritised in future work of TypeCalc, ranked in descending order of priority.

12.3.1 Notable Bugs

Here we provide a list of notable bugs and issues, which should be prioritised in newer versions of TypeCalc, along with references to the sections in which the issue is elaborated on. It should be mentioned that this is not an exhaustive list.

1. Undo/Redo: Undoing cells with dependents does not recalculate the dependents. (Section 12.1.4).
2. Copy/paste does not work for absolute references in the R1C1 notation (Section 7.6).
3. `exportAsXML()`: Export with multiple sheets has invalid format for Excel (Section 6.2.3).
4. Function call evaluation should accommodate newly added functions to Formula.js (Section 12.1.7).
5. `syncScroll()`: Only synchronises grid headers with the grid body, and not vice versa. If trying to synchronise the other way around in the same manner, it will constantly reset the scroll position on the opposite axis (Section 12.1.6).

6. Align undo/redo with the number of actions rather than the number of changed cells (Section 12.1.4).
7. Full recalculation can recalculate a chain of 10,000 cell references starting in A1 and ending at A10000, but not when starting in A10000 and ending in A1 (12.1.3).

13 Conclusion

This report has laid out our thesis project on the spreadsheet implementation TypeCalc, based on the work of Peter Sestoft’s book, *Spreadsheet Implementation Technology* (2014), and his own spreadsheet application, CoreCalc. To broaden our knowledge on the field of spreadsheet implementation, we conducted research into other existing implementations, particularly open-source ones, and identified a gap that we could fill. TypeCalc does this as an open-source spreadsheet implementation written in TypeScript, complete with documentation and a graphical user interface inspired by industry staples like Microsoft Excel and Google Sheets. It sets itself apart further by also employing the specialised QT4 simplified quadtree data structure, providing highly efficient space usage for a spreadsheet application, as explained in Section 7.5.1. While certain aspects remain incomplete, such as the undo/redo feature and the testing suite, as presented in Sections 12.1.4 and 12.2.3, respectively, we ultimately succeeded in most of our goals for TypeCalc, whose design choices and implementation details are specified throughout Sections 6, 7, 8, 9, and 10. There, we elaborate on the API-layer, the back-end, our parser, our implementation of functions, and the front-end, respectively. Additionally, we also achieved all of our own desired learning outcomes from the process, as we argue in 12.2.5. In conclusion, we contribute TypeCalc, an open-source first release of a novel, full-stack spreadsheet application written in TypeScript with intricate implementation details, reflections and documentation for the use of future researchers or developers looking to utilise our work and findings in their own projects.

References

- Aaronsohn, I. (2025). *React-spreadsheet*. <https://github.com/iddan/react-spreadsheet>
- Anthropic. (2025). *Claude*. <https://claude.ai/new>
- Antoun, T., & Zaika, I. (2014). *Bonus talk: C++ in ms office — cppcon*. <https://cppcon.org/bonus-talk-cxx-in-ms-office-2014/>
- Arguelles, C., Ivanković, M., & Bender, A. (2020). *Code coverage best practices*. <https://testing.googleblog.com/2020/08/code-coverage-best-practices.html>
- Asana. (2025). *What is kanban? a beginner's guide for agile teams*. <https://asana.com/resources/what-is-kanban>
- Atlassian. (2025). *Capture, organize, and tackle your to-dos from anywhere — trello*. <https://trello.com/home>
- Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). *Manifesto for agile software development*. <https://agilemanifesto.org/>
- Bender, E. (2004). *Three minutes: Godfathers of the spreadsheet*. <https://web.archive.org/web/20080726090701/http://www.pcworld.com/article/id,116166/article.html>
- Bill, W. (2022). *Integral numeric types*. <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/integral-numeric-types>
- Bill, W. (2024). *Strings*. <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/strings/>
- Bricklin, D. (1999). *Early days*. <http://danbricklin.com/firstspreadsheetquestion.htm>
- Bynens, M. (2017). *Elements kinds in v8*. <https://v8.dev/blog/elements-kinds>
- Chevrotain. (2025a). *Blazing fast*. https://chevrotain.io/docs/features/blazing_fast.html
- Chevrotain. (2025b). *Cst*. https://chevrotain.io/docs/guide/concrete_syntax_tree.html#ast-vs-cst
- Chevrotain. (2025c). *Javascript parsing libraries benchmark*. <https://chevrotain.io/performance/>
- Chinnathambi, K. (2020). *What is ui virtualization?* <https://www.kirupa.com/hodgepodge/ui-virtualization.htm#:~:text=This%20seemingly%20common%20sense%20approach%20is%20known%20as%20C%2C%20DOM%20optimized%20for%20what%20we%20see%20on%20screen.>

- Devero, A. (2021). *Introduction to object types in typescript pt1*. <https://blog.alexdevero.com/object-types-in-typescript-pt1/>
- Docker. (2025). *Docker*. <https://www.docker.com/>
- Duarte, F. (2025). *Google workspace user stats*. <https://explodingtopics.com/blog/google-workspace-stats>
- Duck, B. (2019). *Libreoffice*. https://web.archive.org/web/20190328192305/https://www.openhub.net/p/libreoffice/analyses/latest/languages_summary
- Fallon, N. (2024). *The best spreadsheet software: Features, uses, and programs*. <https://www.business.com/articles/best-spreadsheet-software/>
- Ghalimi, I. C., Karonen, I., Loisel, S., Norris, T., Roönaän, & Stiebitzhofer, H. (2025a). *Formula.js*. <https://formulajs.info/>
- Ghalimi, I. C., Karonen, I., Loisel, S., Norris, T., Roönaän, & Stiebitzhofer, H. (2025b). *Formula.js: Javascript implementation of most microsoft excel formula functions*. <https://github.com/formulajs/formulajs>
- Gift, N., Behrman, K., Deza, A., & Gheorghiu, G. (2019). *Python for devops: Learn ruthlessly effective automation*. O'Reilly.
- Github. (2025a). *Github actions*. <https://github.com/features/actions>
- Github. (2025b). *Npm home*. <https://www.npmjs.com/>
- Goldberg, J. (2025). *Gnumeric github repository*. <https://github.com/GNOME/gnumeric/tree/master>
- Google. (2025). *Measure performance with the rail model*. <https://web.dev/articles/rail>
- Gupta, A. (2025). *Fast-xml-parser*. <https://www.npmjs.com/package/fast-xml-parser>
- Handsontable. (2025a). *Handsontable*. <https://github.com/handsontable/handsontable>
- Handsontable. (2025b). *Hyperformula*. <https://github.com/handsontable/hyperformula/>
- Hejlsberg, A. (2025). *A 10x faster typescript*. <https://devblogs.microsoft.com/typescript/typescript-native-port/>
- JetBrains. (2024). *Software developers statistics 2024 - state of developer ecosystem report*. <https://www.jetbrains.com/lp/devecosystem-2024>
- Kapor, M. (1999). *Kapor email thu, 15 apr 1999* [Archived mail from Mitch Kapor]. <https://www.dssresources.com/history/kapor4151999.html>
- Kimbrough, K. (2025). *Ai is shifting the workplace skillset. but human skills still count*. <https://www.weforum.org/stories/2025/01/ai-workplace-skills/>

- Lyu, L. (2024). *Fastformula*. <https://github.com/LesterLyu/fast-formula-parser>
- MDN. (2025). *Rangerror: Too much recursion*. Mozilla. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Errors/Too_much_recursion
- Merriam-Webster. (2025). *Copy, verb*. <https://www.merriam-webster.com/dictionary/copy>
- Microsoft. (2014). *Xml spreadsheet reference*. [https://learn.microsoft.com/en-us/previous-versions/office/developer/office-xp/aa140066\(v=office.10\)](https://learn.microsoft.com/en-us/previous-versions/office/developer/office-xp/aa140066(v=office.10))
- Microsoft. (2025a). *Casting and type conversions - c#*. <https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/types/casting-and-type-conversions>
- Microsoft. (2025b). *Fill data automatically in worksheet cells*. <https://support.microsoft.com/en-us/office/fill-data-automatically-in-worksheet-cells-74e31bdd-d993-45da-aa82-35a236c5b5db>
- Microsoft. (2025c). *Typescript*. <https://www.typescriptlang.org/>
- Microsoft. (2025d). *Typescript: Documentation - do's and don'ts*. Retrieved June 2, 2025, from <https://www.typescriptlang.org/docs/handbook/declaration-files/do-s-and-don-ts.html#any>
- Mingodad. (2022). *Cocor-typescript*. <https://github.com/mingodad/CocoR-Typescript>
- Mozilla. (2025a). *Flexbox*. https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/CSS_layout/Flexbox
- Mozilla. (2025b). *HTMLTableElement*. <https://developer.mozilla.org/en-US/docs/Web/API/HTMLTableElement>
- Mozilla. (2025c). *Memory management - javascript*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Memory_management
- Mozilla. (2025d). *Window: Localstorage property*. <https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage>
- Mozilla. (2025e). *Window: Sessionstorage property*. <https://developer.mozilla.org/en-US/docs/Web/API/Window/sessionStorage>
- Mozilla Developer Network. (2025a). *Dynamic typing*. https://developer.mozilla.org/en-US/docs/Glossary/Dynamic_typing
- Mozilla Developer Network. (2025b). *Error - javascript — mdn*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error
- Nielsen, J. (1994). *10 usability heuristics for user interface design*. <https://www.nngroup.com/articles/ten-usability-heuristics/>

- Norman, D. A. (2013). *The design of everyday things* (Rev. and expanded edition). MIT press.
- OpenAI. (2025). *Openai*. <https://openai.com/>
- Panchasara, M. (2021). *Typescript vs javascript: Which one is better to choose?* <https://radixweb.com/blog/typescript-vs-javascript>
- Parr, T. (2025). *Antlr*. <https://wwwantlr.org/index.html>
- Playwright. (2025). *Playwright for python*. <https://playwright.dev/python/>
- Prettier. (2025). *Prettier*. <https://prettier.io/>
- React. (2025a). *Describing the ui - react*. React. <https://react.dev/learn/describing-the-ui>
- React. (2025b). *Passing props to a component - react*. React. <https://react.dev/learn/passing-props-to-a-component>
- React. (2025c). *React*. React. <https://react.dev/>
- React. (2025d). *Responding to events - react*. React. <https://react.dev/learn/responding-to-events>
- React. (2025e). *Useeffect - react*. React. <https://react.dev/reference/react/useEffect>
- React. (2025f). *Useref - react*. React. <https://react.dev/reference/react/useRef>
- React. (2025g). *Usestate - react*. React. <https://react.dev/reference/react/useState>
- Sestoft, P. (1999). *Grammars and parsing with java*. <https://studwww.itu.dk/~sestoft/programming/parsernotes.pdf>
- Sestoft, P. (2014a). *Corecalc*. <https://studwww.itu.dk/~sestoft/funcalc/>
- Sestoft, P. (2014b). *Spreadsheet implementation technology: Basics and extensions*. The MIT Press. <https://doi.org/10.7551/mitpress/8647.001.0001>
- Sommerville, I. (2016). *Software engineering* (10th ed.). Pearson Education Limited.
- Syncfusion. (2025). *React spreadsheet*. <https://www.syncfusion.com/react-components/react-spreadsheet>
- Thomas, M., & Steiner, T. (2024). *Why google sheets ported its calculation worker from javascript to wasmgc*. <https://web.dev/case-studies/google-sheets-wasmgc>
- Uddin, A., & Anand, A. (2019). Importance of software testing in the process of software development. *International Journal for Scientific Research and Development*.
- Vaughn, B. (2025a). *React-window*. <https://www.npmjs.com/package/react-window>

- Vaughn, B. (2025b). *React-window - variablesizegrid*. <https://react-window.vercel.app/#/api/VariableSizeGrid>
- Vite. (2025). *Vite*. <https://vite.dev>
- Vitest. (2025). *Vitest — next generation testing framework*. <https://vitest.dev/>
- W3Techs. (2025). *Usage statistics of javascript as client-side programming language on websites*. Retrieved June 2, 2025, from <https://w3techs.com/technologies/details/cp-javascript>
- Williams, M., et al. (2025). *Jsdoc*. <https://jsdoc.app/>