



UNIVERSITEIT VAN AMSTERDAM

Faculteit der Natuurwetenschappen,  
Wiskunde en Informatica

*Master Thesis*

# Excelerate: Towards Compiling Excel to Human Readable Code

Joachim Dekker

*Master Software Engineering*

<b>Student</b>	Joachim Dekker, 15887715, joachim.dekker@{student.uva.nl, infosupport.com}
<b>Academic Supervisor</b>	dr. Andrés Goens (PCS), a.goens@uva.nl
<b>Daily Supervisor</b>	Bjorn Jacobs MSc., bjorn.jacobs@infosupport.com
<b>Host</b>	Info Support B.V., Business Unit Finance (BUF) at Kruisboog 42, 3905 TG Veenendaal
<b>Date</b>	2025-10-01

# Abstract

Microsoft Excel is the most widely used end-user programming platform, yet its formula calculation engine presents a challenge for large scale calculations in the form of long delays and unreliability. This thesis presents *Excelerate*, a novel source-to-source compiler that compiles Excel files to idiomatic C# libraries while preserving complete computational semantics.

The compiler employs a three-phase pipeline: a *Structural Model* parses the Excel Workbook to extract spreadsheets and tables; a *Compute Model* derives the dependency graph of formula compositions; and a *Code Model* emits imperative C# through the use of the Roslyn API. Building on this approach, this thesis introduces *structure-aware compilation* that automatically detects higher-level spreadsheet structures, such as tables and recursive tables. It is able to produce readable, and comprehensive code.

Evaluation is done on five real-world Microsoft Excel templates and demonstrates semantic equality with Excel's native calculation engine. *Excelerate* exhibits a speed-up of 1710x over Excel, indicating substantial performance improvements. These findings show that complex Excel formula compositions within a workbook can be transformed into idiomatic C# code.

# Contents

1. Introduction .....	5
1.1. Problem Description .....	6
1.1.1. Research Questions .....	6
1.1.2. Contributions .....	7
1.2. Related Work .....	7
1.2.1. Function Extraction .....	7
1.2.2. Validation .....	8
1.2.3. Source-to-source compilation .....	8
1.2.4. Idiomatic Code .....	9
1.3. Excel .....	10
1.3.1. Excel Structure .....	10
1.3.2. Formulae .....	12
1.4. Outline .....	13
2. Compiling Excel .....	14
2.1. High Level Overview .....	15
2.2. Extracting the Data .....	17
2.2.1. Structural Model .....	17
2.2.2. Formula Model .....	19
2.2.3. Constructing the model .....	19
2.3. Deriving the Logic .....	22
2.3.1. Compute Model .....	22
2.3.2. Compute Graph .....	23
2.4. Generating Code .....	26
2.4.1. Code Model .....	26
2.4.2. Creating the model .....	28
2.5. Reflecting on the Compiler .....	30
2.5.1. Two Examples .....	30
2.5.2. Missing structure .....	32
2.5.3. Guided by structure .....	32
3. Excelerate .....	34
3.1. Structure-aware compilation .....	35
3.1.1. Types of Structures .....	36
3.2. Changes to the Compiler .....	38
3.2.1. Changes in Input .....	38
3.2.2. Detecting the structures .....	39
3.2.3. Linking the structures .....	39
3.2.4. Converting to code .....	39
3.3. Finding the Structures .....	40
3.3.1. Changes in the model .....	40
3.3.2. Areas .....	40
3.3.3. Classifying Structures .....	42
3.4. Embedding the Structures .....	45
3.4.1. Compute Grid .....	45
3.4.2. Compute Model .....	46
3.4.3. Changes in the Compiler Pipeline .....	47
3.5. Coding the Structures .....	49

3.5.1. Creating types from structures .....	49
3.5.2. Optimisations for Mutual Recursion .....	51
3.6. Discussion of the Compiler Design .....	52
3.6.1. Data Model .....	52
3.6.2. Computed Properties .....	52
4. Evaluation .....	54
4.1. Methods .....	55
4.1.1. Semantics and Performance .....	55
4.1.2. Readability .....	58
4.2. Results .....	59
4.2.1. Semantic Equality .....	59
4.2.2. Performance .....	59
4.3. Discussion .....	61
4.3.1. Readability and Idiomaticity .....	62
4.3.2. Threats to validity .....	65
5. Conclusion .....	66
5.1. Conclusion .....	67
5.1.1. Mapping Excel formulae to human-readable C# code .....	67
5.1.2. Common Excel Structures .....	67
5.1.3. Verification .....	67
5.1.4. Performance differences .....	68
5.2. Future Work .....	68
Bibliography .....	69
A. Full code samples .....	73

*Chapter 1*  
**Introduction**

**1.1. Problem Description ..... 6**  
**1.2. Related Work ..... 7**  
**1.3. Excel ..... 10**  
**1.4. Outline ..... 13**



Microsoft Excel is arguably the most widely used programming environment worldwide [1]. The interface can be seen as the IDE, and the worksheet and the formulae as the code. At some pension fund companies, it is reportedly<sup>1</sup> used to maintain and calculate the pension fund interest for all of its customers.

Pension fund employees use Excel worksheets as their tool to compute the forecasts under a range of different situations. There are more than a thousand situations per person. For example, a user might want to know what their pension will look like if they take their pension when 65 years old, given the economy has been good. The pension fund calculates the prognosis for this with Excel. As such, Excel is used like an API that calculates these situations.

For some companies, all of these situations need to be calculated for nearly 1 million customers every year. Due to the slow nature of Excel, this process usually takes weeks, encountering many crashes that slow down progress even further. To combat this, one of the partners of the pension funds, Info Support B.V., manually converted such an Excel file to a high-level programming language in order to increase performance and stability.

However, the laws surrounding pensions change every year [2]. As a result, ways to calculate the aforementioned situations need to be changed. Since pension fund employees are unable to maintain the code provided by Info Support, after every change, this needs to be manually changed by Info Support. This resource-intensive operation could be automated. Hence, Info Support—as part of the ongoing *GROENpensioen* project<sup>2</sup>—aims to automatically convert these calculations from Microsoft Excel to high performing code to improve reliability, performance and ultimately durability of the pension fund calculation process.

## 1.1. Problem Description

In this thesis, we refer to the Excel compiler as a tool that accepts an Excel file together with sets of input and output cells, and produces a self-contained class library. This library is then to be used in external tools and computes the same outputs for any given inputs as the original spreadsheet.

Automatically converting Excel workbooks to high-performing, readable code is not straightforward. Excel sheets are often large and result in large compositions of formulae, which in turn result in complex semantics. The desired Excel compiler should support these formulae and the different execution paths these Excel workbooks have to offer. Furthermore, Excel exhibits complex semantics with cyclic dependencies, dynamic range operators, and array formulas.

Human auditors must be able to inspect the program. Besides it needing to be semantically equivalent, it needs to be readable, using idiomatic code. The auditor should be able to easily understand the code instead of deciphering bytecode. Given that the spreadsheet structure does not easily compile to imperative code, a big challenge tackled in this thesis is understanding the semantics of a spreadsheet and converting it to imperative code.

### 1.1.1. Research Questions

To address these problems, we utilise an exploratory study with design elements. Within the study, we propose the following research questions to guide the design process:

**(RQ1)** How can complex Excel formula compositions be mapped to human-readable C# code?

---

<sup>1</sup>Based on several interviews with employees at Info Support B.V.

<sup>2</sup>The graduation project is called ‘Van GRIJSPensioen naar GROENpensioen’ and can be found at <https://carriere.infosupport.com/groenpensioen/>

- (RQ2) What are common excel structures and how can they be applied to optimise the compilation of excel formula compositions?
- (RQ3) What are the performance differences between EXCELERATE and Excel?
- (RQ4) How can the mapping between excel formulas and code be verified?

The relevance of these research questions have been made clear in the previous sections. The main research question (RQ1) is the main goal of the compiler design, since we need to find a way to convert the Excel files to code in a idiomatic way. The second research question (RQ2) helps with this, providing structures and seeing if they can help guide the compiler. The last two research questions are the evaluation of the compiler, searching for performance improvements and semantic equality.

## 1.1.2. Contributions

Within this thesis, we make theoretical and practical contributions. First, we introduce a novel three-phase compiler design that compiles Microsoft Excel workbooks into human-readable, idiomatic C# libraries. The compiler is structured with three segregated phases: the *Structural Phase* that parses worksheets, the *Compute Phase* that captures the dependency graph, and the *Code Phase* that produces idiomatic code and emits C# libraries.

Furthermore, we propose *structure-aware compilation*. This approach detects and embeds higher-level structures from the spreadsheets into the dependency graph. We present and implement two structures in this thesis: the table and the chain. Using *structure-aware compilation*, we avoid duplication and improve readability and comprehensibility.

Finally, we use a differential verification method to compare EXCELERATE with Microsoft Excel using randomised input sets. We compare semantic equality, performance and readability across five real-world workbooks. Ultimately, we show that EXCELERATE significantly outperforms Excel, even for large workbooks.

## 1.2. Related Work

As far as we know, this is the first work that compiles Excel to another source language. In this section, we present related work that could help this thesis. We first discuss function extraction, which extract the computational model from Excel to another representation. Then, we discuss ways to validate Excel. We cover different techniques of source-to-source compilation, and finally discuss what it means to have ‘idiomatic’ code.

### 1.2.1. Function Extraction

K. Lano *et al.* [3] describe a way to convert an excel application to code by extracting a UML diagram out of a spreadsheet and converting it to code, but the whole process is manual.

Object oriented models are often used to validate spreadsheets to reduce errors [4], [5]. The automated extraction of such a model is also proposed by J. Cunha *et al.* [6], where functional dependencies are used to detect dependencies between columns like in database normalisation, and construct a model of this. They augment this with OCL to also describe the models constraints [7]. In relation to this research, these works see spreadsheets as databases with relational mappings, while our research only considers the semantic computational model of Excel.

P. Sestoft [8] describes a full alternative implementation of spreadsheets, with subsequent master theses expanding upon the work [9], [10] describe how to efficiently calculate the values in a spreadsheet using the support graph: a more organised version of the functional dependencies used in [6]. In

comparison with J. Cunha *et al.* [6], these works also consider transitive dependencies in their model. Furthermore, they only represent the calculations, but do not generate code except for [9]. T. S. Iversen [9] expands upon the work of P. Sestoft [8] by introducing runtime code generation for speeding up parts of the calculations.

Outside the literature, few packages exist that support calling excel calculations. EPPlus [11], Espose Cells [12], Apache POI [13], and SyncFusion [14] all use their own calculation engine based on the dependencies, but do not use a support graph like in [8]. These tools use the same interpreting technique as Excel, and do not utilise source code generation like this thesis.

## 1.2.2. Validation

To verify the semantics of the program, the semantics of Excel should be defined so they can be compared with the semantics of the higher level language.

A. A. Bock *et al.* [15] defines the operational semantics for a self-made spreadsheet framework [16] which closely reflects and closely resembles the Excel semantics. The semantics could serve as a starting point for verifying the compiler steps from Excel formulas to generated code, but need alteration because it is not clear if they fully model the excel semantics.

D. Steinhöfel *et al.* [17] describes validation of source-to-source compilation using symbolic execution, where the program is ‘executed’ using inference rules to create a symbolic execution tree that can be used to compare the two sources. If their symbolic execution tree is equivalent, the programs are considered equivalent [17]. While the formal verification of the compiler is a good idea, we acknowledge that this is too big for the scope of this master project.

A more manageable verification is empirical verification. G. Rothermel *et al.* [18] and M. Fisher *et al.* [19] introduce the *What You See Is What You Test* (WYSIWYT) framework for spreadsheet testing which sets it apart as empirical verification for spreadsheets. It utilises definition-use (du) associations to link formulas to their computational or predicate uses. A spreadsheet is considered ‘validated’ when all du-associations are exercised by at least one test. While users can manually validate du-associations [18], the framework also supports generating automated test cases using random or goal-oriented approaches to satisfy all du-associations [19]. This idea can be applied to our research. For a good compilation, the generated code should pass all of these automated tests to be sufficiently verified.

## 1.2.3. Source-to-source compilation

Source to source compilation, also called transpilation, transcompilation [20], or program extraction, involves the translation of one higher-level source language to a target higher-level language. The compilation should, like a normal compiler, preserve the semantics. Several techniques have been implemented.

R. Waters [21] and D. Ordóñez Camacho *et al.* [22] both describe *transliteration and refinement*, as well as *abstraction and reimplementatation* [21]. Transliteration involves the translation of code line-by-line to a intermediate translated representation in the target language [21]. The intermediate representation can then be refined to better source code [21]. [22] translate grammars through transliteration, converting one language to the other using simple rules. They allow the user to define exceptions, which serve the same role, albeit a bit more expressive than refinement. J. Cockx *et al.* [23] use simple rules to translate Agda programs into Haskell programs, essentially only doing the transliteration step. Transliteration can be applied in our research, but it mostly works with source code that is alike in grammar and semantics [21].

*Abstraction and reimplementaion* is a more used method [21], [22], [24]. From the source language, an intermediary representation is extracted, such as ILR [22] or UML+OCL [25]. The intermediary representation is then converted to the target language [21]. This method only needs  $2n$  instead of  $n(n-1)$  translators [22], [24]. This closely relates to our research, since we will be implementing a custom domain model that models the excel workbook.

More recently, models trained on large corpora have become a dominant approach for source-to-source compilation [20], [26], [27]. These methods leverage the ability of LLMs to generalize across tasks without specific re-training and provide guidance in their shortcomings [26]. Z. Yang *et al.* [26] propose using test case generation to provide hints to repair incorrect translations. However, these models require lots of annotated data [20]. Hence, B. Roziere *et al.* [20] proposes a way to train a model with unsupervised data, by using large corpora of open source code and applying transformations on them. D. Guo *et al.* [28] and X. Chen *et al.* [27] propose using the AST of the source code as an additional heuristic in transforming programs to reduce errors and improve translation. Since this novel approach increases readability, it does relate to our research. However, given the lack of relevant data, we think the method will be insufficient.

## 1.2.4. Idiomatic Code

*Idiomatic* means “containing expressions that are natural to a native speaker of a language” [29]. While this definition is meant for natural languages, we can also extend it to programming languages: “containing expressions that are natural to a senior programmer writing the language”. Software engineers consider creating code as a big part of their job [30], but one universal definition of what is considered ‘good’ code is hard to determine. This was made clear by J. Börstler *et al.* [31] in their paper—aptly titled “I know it when I see it”—indicating that factors of code quality are diverse across demographics [31].

Yet, a plethora of books and papers have been written on what is considered good or idiomatic code [32]–[36]. J. Börstler *et al.* [37] interviewed developers, students and educators on the factors of code quality. Comprehensibility and Structure were the most commonly named factors [31], [37], which is confirmed in several other sources [32]–[36]. According to M. Fowler *et al.* [34], structure directly relates to comprehensibility and readability since humans read code: if the code does not have structure, it will be harder to read and understand. They give an example where an extremely long function is harder to read than multiple shorter functions with one orchestrating function.

An important factor of structure is no duplication—commonly stated as DRY or ‘Don’t Repeat Yourself’ [34], [36], [37]. Duplication can increase risks when a part of the duplicated code requires rework: often the duplicated part is forgotten and does not get updated [34], [36]. For example, when applying an operation on a list, it is very uncommon (and a bad code practice) to verbosely apply the operation to every array element. Instead, we use a for loop or map that loops over the list and applies the operation.

All of these factors contribute to one larger code quality: readability [31], [33], [37]. According to S. Fakhoury *et al.* [38], readability influences cognitive load and performance of developers. Comprehensibility and structure are very important [31], [37]. Combined with reduced duplication it leads to better readability, which is something we want to achieve.

More concretely: in this thesis, we consider ‘good’ code as code a senior developer would have written in their best language. Furthermore, we follow the style guide of the language, written by Microsoft [39]. This style guide establishes that we need to use the newest language features, write clear, concise code, and adhere to the naming conventions. We acknowledge this is still vague and that idiomatic code is pretty subjective. We will give more examples of what we consider good C# code in Section 4.3.1.

## 1.3. Excel

Excel is an (online) spreadsheet application developed by Microsoft. It is used extensively in the world of Finance, as can be inferred by the sheer amount of financial-themed books and papers on the topic. It allows for data modelling, manipulation and analysis, transforming a painful manual calculation into an automated process in just a few clicks.

In this subsection, we cover the basic principles of Excel necessary to understand this thesis. First, we discuss the structure of an *Excel Workbook*. Then, we look at Excel formulae. Finally, we give a high-level overview of the calculations in Excel.

### 1.3.1. Excel Structure

All information of an Excel file is stored in an *Excel Workbook*. The workbook is the aggregate of all excel entities, containing standard styles, file configurations, preferences, and most importantly, the *Excel Worksheets*.

Historically, the *Excel Workbook* file has been housed under the `.xls` extension. However, since Excel 2007 the Excel workbook is saved as an `.xlsx` file: a zip compressed folder comprised of XML files. These files contain information on metadata, or contain the actual data and formulae of the worksheets.

The real data is stored in the *Excel Worksheet*. This spreadsheet is a two-by-two grid of cells, which can be filled with values. The values in a cell can also be calculated, but the user will only see the value unless clicking on the cell. Besides the cell-structure, the Excel worksheet can also contain graphs and images. Furthermore, the cells can be styled in a certain format, and will follow the standard style of the workbook by default. That said, in this thesis, we mostly look at the data and the computations on the data.

Excel also has other *special structures*. These structures live inside the grid, but augment the functionality of excel, simplifying certain operations.

### Example: Family Budget

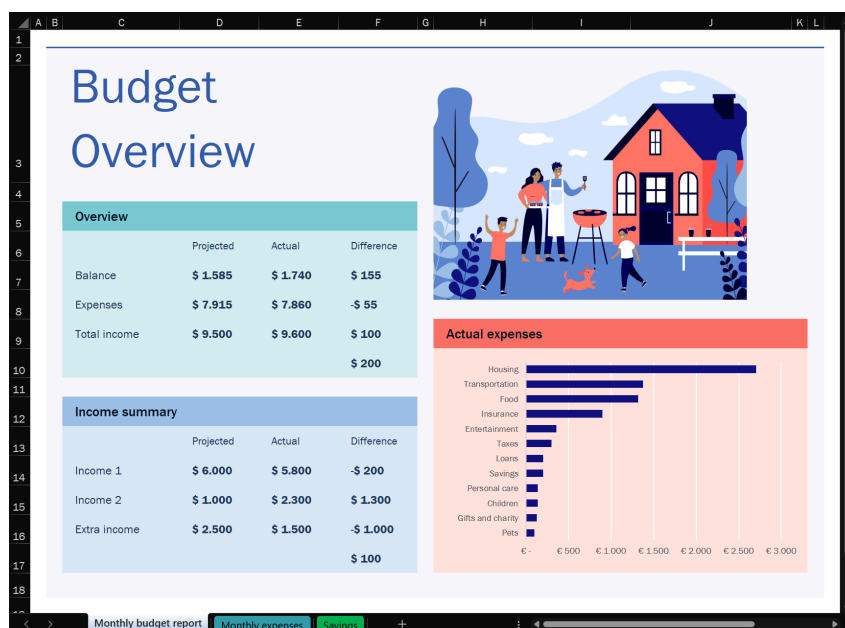


Figure 1.1: The *Monthly Budget Report* overview spreadsheet of the *Family Budget* workbook.

A good example is the *Family Budget* workbook, which mimics a budgeting application. In this workbook, we can see three are three *worksheets*, namely the *Monthly budget report* (which is selected and is visible in Figure 1.1), *Monthly expenses*, and *Savings*. Figure 1.1 also shows the styling that is possible in the Excel sheet. This styling is also saved in the workbook.

### Tables

*Excel Tables* elevate normal spreadsheets by introducing functionality unique to Excel Tables [40]. The table structure is arguably the most common structure in Excel, where a table is created with columns of a certain type, and rows filled with information. With Excel tables, it is possible to formalise these notions, and explicitly define columns and the size of the table.

*Excel Tables* have a name. Their columns also have a name. It is possible to use these names in a *structured reference*, where instead of defining the range of the column like `A1:A50`, a more semantic notation `tableName[columnName]` is used. This notation is just like indexing an array or a dictionary, and provides more readable formulae [40].

There are two types of columns in an Excel table. Columns can have arbitrary data and computations, or have a fixed, computed formula for all rows in the column [40]. This formula can reference other columns using a special notation. For example, a column can reference other columns by using the `@` notation in a *structured reference*, resulting in something like `@[column1] + @[column2] * $A$1`. Every row in the table will then update the column with that formula, and the `@`-notation will automatically target the cell in the row and specified column.

For example, in Figure 1.2 the monthly expenses table is a designated table with five columns. The table has four data columns, and one computed column. In the difference column, the formula in the upper row is `Monthly expenses'!$E5-'Monthly expenses'!$F5`. In the background, Excel recognises the dependencies in this formula and automatically applies transformations to make it work in the other rows. So for example, in row seven, the formula is changed to `Monthly expenses'!$E5-'Monthly expenses'!$F5`.

Description	Category	Projected cost	Actual cost	Difference
Extracurricular activities	Children	\$ 40.00	\$ 40.00	-
Medical	Children	\$ -	\$ -	-
School supplies	Children	\$ -	\$ -	-
School tuition	Children	\$ 100.00	\$ 100.00	-
Concerts	Entertainment	\$ 50.00	\$ 40.00	\$ 10.00
Live theater	Entertainment	\$ 200.00	\$ 150.00	\$ 50.00
Movies	Entertainment	\$ 50.00	\$ 28.00	\$ 22.00
Music (CDs, downloads, etc.)	Entertainment	\$ 50.00	\$ 30.00	\$ 20.00
Sporting events	Entertainment	\$ -	\$ 40.00	\$ (40.00)
Video (purchase)	Entertainment	\$ 20.00	\$ 50.00	\$ (30.00)
Video (rental)	Entertainment	\$ 30.00	\$ 20.00	\$ 10.00
Dining out	Food	\$ 1,000.00	\$ 1,200.00	\$ (200.00)
Groceries	Food	\$ 100.00	\$ 120.00	\$ (20.00)
Charity 1	Gifts and charity	\$ 75.00	\$ 100.00	\$ (25.00)
Charity 2	Gifts and charity	\$ 25.00	\$ 25.00	-

Figure 1.2: The *Monthly expenses* spreadsheet in the *Family Budget* workbook: A table with five columns describing the columns. The *Difference* column is computed from the actual cost and projected cost column.

## PivotTables

The *Excel PivotTable* is a powerful tool for summarising and analysing data from external sources [41]. Unlike normal tables, *PivotTables* are dynamic in layout, and allow for reorganisation of the data. It provides a dynamic view of the underlying data source and allows the user to perform operations on the data [41]. The core feature of *PivotTables* lies in their ability to perform aggregations as a low-code solution. Without writing a single formula, it is possible to construct comprehensive views of the data. This is made even easier with automatic refreshing, which updates the PivotTable when there is a change in the underlying data model [41].

*PivotTables* support grouping. Dates, for example, can be grouped into months or years, and numeric data can be binned into intervals. These groupings are handled internally and allow for a more compact and meaningful summary [41].

It is important to note that PivotTables do not compute data themselves [41]. Instead, they act as a kind of presentation layer of other datasources. These datasources can be internal or external. Hence, while they could depend on the data and formulas defined elsewhere, they do not themselves perform cell-level computations in the same way as formulas in worksheets or structured references in *Excel Tables* [41].

### 1.3.2. Formulae

In Excel, a formula is an expression that calculates the value of a cell [42]. Every formula begins with an equal sign =, which tells Excel that the contents of the cell constitute a formula [42]. These formulas can contain functions, references, operators, and constants to perform calculations on data within the Excel sheet. These distinct elements all contribute different functionality to the end results:

- Constants are fixed values entered directly into the formula.
- Functions are predefined operations that perform specific calculations on their arguments. For example, `SUM(1;2;3;4;5)` will sum up the arguments. An interesting thing with Excel is that the names and notation of the functions are language dependent [42]. For instance, in Dutch, the `SUM` function is translated to `SOM`. As well as the notation for the arguments list, where in one language the arguments are split with a semi-colon ;, another language splits them with a comma ,.
- Operators are special functions that can be inlined. For example, the plus + or minus - operators. For the order of operations, Excel follows the PEMDAS rule [42].
- References provide pointers to other cells or ranges within the worksheet in the form of the *A1-notation*. This notation targets the rows and columns, where the columns are represented by the alphabet, and the rows by a number. For instance, the cell in the third column, and fourth row, would be `C4`. References can be *relative* to the cell or *absolute*. *absolute references* are started with a dollar sign \$. It is possible to only make the row or column absolute, to create a *hybrid reference*.
  - It is possible to reference other worksheets by appending the worksheet name to the reference with an '!' infix. For example, to reference cell `C4` in the `Monthly Expenses` from anywhere in the workbook, we use `'Monthly Expenses'!C4`. Notice how the use of spaces is allowed as long as the name of the worksheet is between quotes ( ' ' ).

## 1.4. Outline

After the background on the literature and Excel, we can start discussing the Excel compilers we introduce in this thesis. The remainder of this thesis is organised as follows.

In Chapter 2, we introduce a ‘basic’ Excel compiler and explain the three phases (Structural, Compute, and Code) that transform an Excel workbook into a C# library. We show that the compiler produces correct code, but discuss limitations in readability.

Hoping to improve this readability, we introduce a more advanced compiler EXCELERATE and the concept of *Structure-aware Compilation* in Chapter 3. The compiler builds upon the ‘basic’ compiler. This chapter details the *Table* and *Chain* structures and covers how they are integrated into the ‘basic’ compiler.

Chapter 4 describes the methodology and results for our experiments that evaluate EXCELERATE and Excel using differential verification. We assess semantic equality, benchmark performance and evaluate the readability of the emitted code.

Finally, Chapter 5 concludes the thesis, summarising the main findings and contributions, answering the research questions and outlines potential improvements for further work.

*Chapter 2*  
**Compiling Excel**

2.1. High Level Overview . . . . 15  
2.2. Extracting the Data . . . . . 17  
2.3. Deriving the Logic . . . . . 22  
2.4. Generating Code . . . . . 26  
2.5. Reflecting on the  
Compiler . . . . . 30



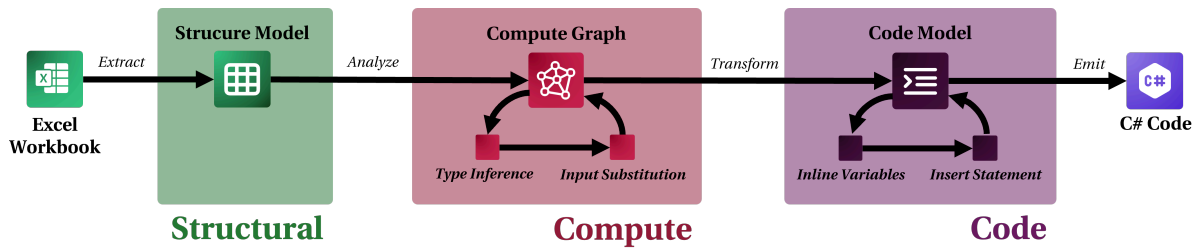


Figure 2.1: The flow of the basic compiler. The content of the Excel workbook is extracted and parsed into the *Structure Model* (Workbook). Then this workbook is further analysed to find the underlying compute model and is transformed into a compute grid. The disconnected cells are then connected through their formula dependencies into a compute graph. Finally, a generic code layout model is created and C# code is emitted through the Roslyn API.

The first step in creating human readable code and answering the research questions is to convert the Excel sheet to executable code. Spreadsheets are executable programs at their core. The web of formulae, linked through dependencies forms the execution model of the spreadsheet. To translate this model into human-readable, verifiable, and reusable code, we first need a mechanism that can compile an Excel workbook into a general-purpose language.

This chapter introduces such a mechanism: the ‘basic’ Excel compiler. This straightforward compiler’s sole task is to transform any spreadsheet into executable C# code. As we will see, we require three separate phases for this: extracting relevant data in the *Structure Phase*; constructing a computational plan in the *Compute Phase*; and emitting semantic-preserving C# code in the *Code Phase*.

We start with a high level overview of the compiler, briefly touching upon each phase. Then, we dive deeper into each phase: discussing the intermediate representations used, and the transformation step needed to compile Excel. Finally, we demonstrate the compiler at the end of the chapter, discussing its shortcomings and setting the stage for a more advanced technique which will be the focus of the following chapter.

## 2.1. High Level Overview

Compiling an Excel Spreadsheet to C# code is done in several steps. Before we dive deeper into the different compiler steps shown in Figure 2.1, we first give a high level overview of the compiler in order to get a feel for what is coming. We begin with a `.xlsx` spreadsheet file, the standard file format for Excel files. As an example, consider Spreadsheet 2.1, a small spreadsheet that has been saved as `spreadsheet.xlsx`. This spreadsheet will be compiled into a program using the compiler. We will use cell `D5` as an output cell, meaning that the compiled code should output the sum of the differences.

In the first phase of the compiler, the *Structural Phase*, we extract the *Structural Model* or *Workbook* out of the provided spreadsheet. This is comparable with getting an AST of a code file: we get an

	A	B	C	D
1		<b>Projected</b>	<b>Actual</b>	<b>Difference</b>
2	Income 1	6.000	5.800	=C2-B2
3	Income 2	1.000	2.300	=C3-B3
4	Extra Income	2.500	1.500	=C4-B4
5	TOTAL			=SUM(D2:D4)

Spreadsheet 2.1: A extract of the ‘Family monthly budget’ sheet in the ‘Family monthly budget’. This spreadsheet calculates the differences in projected and actual income.

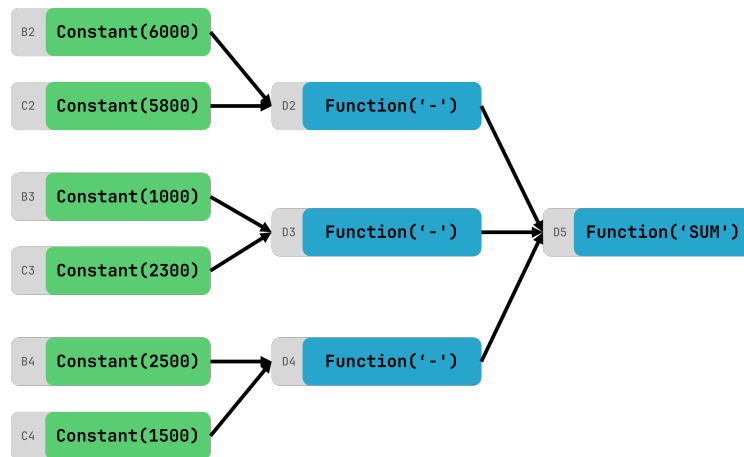


Figure 2.2: The compute graph generated from the formulae and dependencies.

intermediate language that models the spreadsheet and allows for easy manipulation and analysis. As such, we extract specific information from the whole Excel file: cells, formulae, tables. This is done through reading the XML files embedded in the `spreadsheet.xlsx` and parsing them to an internal model.

In the *Compute Phase*, the next phase of the compiler, we construct a *Compute Graph*. This graph models the computational flow in the Excel file. Based on the provided output—in this case `D5`—we analyse the formulae in the provided cell and convert it to the *Compute Model*. The dependencies of the formula are located using the *Structural Model* and recursively converted to this new model. The result is a graph of formulae that feed into each other and ultimately compute the value of the output cell. For Spreadsheet 2.1 the result can be found in Figure 2.2.

Then we enter the *Code Phase* of the compiler. This phase converts the *Compute Graph* created in the previous phase to the *Code Model*. This intermediate representation is specifically created for easy manipulation and is meant to be language agnostic, supporting both functional as imperative paradigms. In this phase, we inline variables and transform the code to be more in line with C#'s imperative nature. Both these compiler passes have the purpose of simplifying the code.

Lastly, the data in the *Code Model* is transformed using the C# compiler platform API (Roslyn) to C# code classes. The class library is created and is now ready for use in external scripts. In Listing 2.1, the final compiled version of Spreadsheet 2.1 can be seen that was compiled using this 'basic' Excel compiler.

Now that we have given a grand overview, we will dive deeper into the different phases of the compiler. We first discuss the *Structural Phase*, extracting the spreadsheet data. Then, we move over to the *Compute Phase*, formally describing the compute model in the form of the *Compute Graph*. Finally, we explain the *Code Phase* and the step from the *Code Model* to the actual C# code.

```

public class Program
{
    public double Main()
    {
        double sheet1D2 = 5800 - 6000;
        double sheet1D3 = 2300 - 1000;
        double sheet1D4 = 1500 - 2500;
        double sheet1D5 = new List<double> { sheet1D2, sheet1D3, sheet1D4 }.Sum();
        return sheet1D5;
    }
}
  
```

Listing 2.1: A compiled version of Spreadsheet 2.1 with the 'basic' Excel compiler described in Section 2.1. The compiler emits a method that calculates the value of the `D5` cell.

```

record Workbook(string Name, Table[] Tables, Spreadsheet[] Spreadsheets);

record Spreadsheet(string Name, Set<Cell> Cells);

record Cell(Location Location);
  → record EmptyCell();
  → record ValueCell<T>(T Value);
  → record FormulaCell(Formula Formula);

record Table(string Name, Range Location, string[] Columns);

```

Listing 2.2: The formal definition of the Structure Model. It essentially represents the Excel workbook. A record represents a node in the AST. Every record has properties. The AST is built-up from the Typed properties in records: if one record has another record as property, they are linked in the AST.

## 2.2. Extracting the Data

In order to analyse the Excel file and its spreadsheets, we need to parse it to our internal data structure. In this section, we introduce the *Structural Model*: an internal data structure mimics the objects found in Excel we covered in Section 1.3. The *Structural Phase* contains the parsing of an Excel file to the *Structural Model*. In the next chapter, we will expand this phase to detect structures in the spreadsheet and enhance the *Structural Model*.

The *Structural Model* only parses the Excel file and as such does not work with the input and output the user has defined. This has two reasons. The primary reason is that we need the AST of the spreadsheet to analyse in which area we are. Besides, we could prune certain ‘unused’ sections of the spreadsheet while creating the AST, but makes this phase unnecessary complex. For simplicity, we load the entire spreadsheet and prune unused parts in the next phases.

In this section, we will first discuss what the *Structural Model* entails and introduce the *Formula Model*: a simplified AST for Excel formulae. Then, we discuss what an Excel file contains and how the *Structural Model* is extracted from the Excel Worksheet.

### 2.2.1. Structural Model

The *Structural Model* is a data structure that captures the contents of an Excel file and allows for manipulation and analysis on cell level. The model preserves the tabular format of Excel spreadsheets. Listing 2.2 describes the formal definition of the *Structural Model*.

#### Workbook

The *Workbook* serves as the top-level model of the Excel file. As such, it is the top node of our AST. It encapsulates all the data and structure. A workbook contains references to one or more *Spreadsheets*. Additionally, *Excel Tables* that are created in *Spreadsheets* are also stored at this level. This is to accommodate simpler look-up for tables, which can be referenced from any *Spreadsheet* in the *Workbook*. Furthermore, a workbook has an associated name, which usually mirrors the filename of the original Excel file.

#### Spreadsheet

The *spreadsheet* is derived from the *Excel Worksheet* (See Section 1.3): it is sparse representation of a two-dimensional grid of cells. We only store the non-empty cells. When a location is not found within the data structure, we return an *Empty Cell*. Additionally, the *Spreadsheet* is named in order to support inter-*Spreadsheet* references (see Section 1.3).

#### Cell

The *Cell* is the atomic unit in a spreadsheet. A cell is represented by a location and a value. Depending on the content of the value, it represents a constant value or computation. If the content is a raw value like numbers or text, we say that the cell is a *Value Cell*. If the content is a formula expression that starts with `=`, we call the cell a *Formula Cell*.

**Value Cell.** The *Value Cell* is a cell that contains a raw value. A *Value Cell* will always have a type that can be determined at the time we read the value. For instance, cell `B1` in Spreadsheet 2.1 is a *String Value Cell*, while `C3` is a *Numeric Value Cell*. Excel stores all numbers in the same generic numeric format. To stay consistent, the compiler also does not distinguish between different numerical types.

The values of cells in Excel can be formatted to provide extra visual context for the Excel user. For instance, a number 4.00 can be formatted to let it look like a currency €4.00. It is often used with dates. Within the *Structural Model*, we do not consider value formatting since it does not alter the calculations. Furthermore, Excel stores the original value of a cell apart from the formatted value, and as such, we can always use this normalised value.

**Formula Cell.** A *Formula Cell* is a cell that contains a *Formula*. Just like we discussed for Excel in Section 1.3, the *Formula* calculates a value using dependencies and functions available in the *Workbook*. The *Formula* is immediately parsed to the *Formula Model* in order to access the AST. This also allows for analysis in the next phase. We discuss the *Formula Model* in the next subsection.

## Table

All *Excel Tables*, as described in Section 1.3, are mapped to a *Table*. A *Table* is a sub-section of the grid in the *Spreadsheet* describing data in a tabular format. Like *Excel Tables*, *Tables* cannot overlap, i.e. two *Tables* cannot share columns. *Tables* have a name and defined columns. The columns may have a header giving them a descriptive name, such as ‘Description’ and ‘Category’ in the ‘Monthly Family Budget’ *Excel Worksheets*.

## References

An important construct we have not talked about is the reference. References are a way to refer to a *Cell* or a *Range*. They link cells together by representing the (computed) value in another cell or range.

While we discuss them here, we consider them as their own. They are also utilised in other phases of the compiler to refer to the original location on the spreadsheet.

We distinguish between:

- *Location*: references to cells like `B3` ;
- *Range*: references to a continuous selection of cells in the grid of the *Spreadsheet*, like `A1:B3` ;
- *Table Reference*: references to a column in a table like `Table[ColumnName]` .

We distinguish between these references since they contain information that we can use in the next phases. Take the cell `B3` for instance. This cell reference is essentially the same as `B3:B3` as they both point to the same cell. However, the `B3` cell reference is conceptually different than `B3:B3` in programming terms, as the cell reference is just a single value, and the `B3:B3` is a singleton array.

```
struct Reference();
  → struct Location(int Row, int Column, string Spreadsheet)
  → struct Range(Location From, Location To)
  → struct TableReference(string Table, string[] Columns)
```

Listing 2.3: The formal definition of the References. These models are used throughout the compiler and thus stand apart from the *Structural Model*. A record represents a node in the AST. Every record has properties, which can be other AST nodes (record properties) or metadata (struct properties).

```

record FormulaExpression();
→ record Function(string name, FormulaExpression[] arguments)
→ record Constant(object value)
→ record Reference()
→ record Cell(Location location)
→ record Range(References.Range range)
→ record Table(TableReference tableReferences)

```

Listing 2.4: The formal definition of the Formula Model. It models the Excel Formula Language. A record represents a node in the AST. Every record has properties, which can be other AST nodes (record properties) or metadata (struct properties). Since there is a duplicate name ‘Range’, we added a namespace to the external entities. ‘References.Range’ refers to the *Range* discussed in the previous section.

## 2.2.2. Formula Model

The *Formula Model* models the Excel Formula language. It is a simple DSL that allows us to capture the functions, constant and references used in the formulae in a cell. The *Formula Model* is a Intermediate Representation In this subsection, we briefly discuss the formula model. As we will cover in Section 2.3, the Formula Model was the inspiration for the *Compute Model*. As such, many elements will look the same. The formal definition can be found in Listing 2.4

All entities we cover in the next sections are *Formula Expressions*. A formula is always one big *Formula Expression* that can be composed of several sub-expression. For instance, the `=SUM(A1:B3)` can be converted to the `Function("SUM", Range(A1:B3))` *Formula Expression*, which in turn contains the `Range(A1:B3)` sub-expression.

### Function

The *Function* represents a named procedure with optional parameters. It takes other *Formula Expressions* as input and computes a value as output.

**Operator.** An *Operator* is a special function. Just like a function it has a name and takes parameters to compute a value. However, it is fixed between operands. A common example of an operator is the `+` operator, which is placed between two operands like `1+1`

### Constant

A *Constant* is a value of any type that is used as dependencies in other *Formula Expressions*.

### Reference

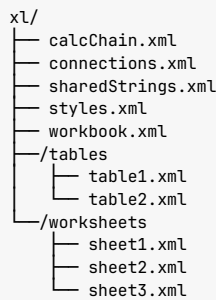
The *Reference* uses the references discussed in the previous section to reference values calculated by *formulae* in other cells. We consider three self-explanatory types: *Cell References*, *Range References* and *Table References*.

With the *Structural Model* and *Formula Model* defined, we have made it possible to describe full Excel files. Within this phase we do not have any other compiler steps. In the next chapter, we will analyse the whole *Worksheet* for so-called *Structures*. These *Structures* enrich the *Structural Model* and go beyond plainly copying the Excel sheet.

Before we move to the *Compute Phase*, we first want to discuss how we managed to convert the Excel Workbook to the *Structure Model*.

## 2.2.3. Constructing the model

Like we discussed, an Excel File contains all information about the workbook and the spreadsheets within. In the previous sections, we already discussed what we extract and what the importance of



Listing 2.5: The structure of an `.xlsx` file. The tree has been slightly edited to improve legibility: folders like `media`, `drawings` and printer settings, as well as other XML configuration files have been removed.

the extracted content was. That said, we did not cover how we obtain this information and how we actually construct the model. In this subsection, we clarify this omission.

## Structure of an Excel File

An Excel file is essentially an archive with individual files describing the *Excel Workbook*. Older versions of Excel (before Excel 2007) create `.xls` files that were harder to read due to the encoded nature of the files. Luckily, the newer `.xlsx` format is an archive consisting of XML files that is easily readable. These files use the `SpreadsheetML` spreadsheet dialect [43] to describe the workbook.

Due to the encoded nature of the older Excel files, the proposed compiler only supports `.xlsx` files. These files are easier to read and convert to the *Structure Model*. However, since the *Structure Model* describes an abstract version of the spreadsheet, future work could include a parser for `.xls` files.

The `.xlsx` archive presented in Listing 2.5 provides an overview of what can be found in an Excel file. Many files are not used, such as:

- `connections.xml` storing connections to outside sources;
- `sharedStrings.xml` storing all unique strings in the document, which is used for space preserving;
- `styles.xml` containing all styles in the workbook.

We also do not use `calcChain.xml`, which contains the order of operations for calculating all cells with formulae in a workbook. This may be counterintuitive, since it could be valuable information for compilation. However, as we will see in the next chapter, we use optimisations that change the order of operations.

We will use `workbook.xml`, which references the XML files in the `tables` and `worksheets` directories. Ultimately, these files build up the *Structure Model*. The `workbook.xml` contains metadata like the name of the workbook. In the `tables` directory, we find the files that describe the tables. Such a file contains the names and columns of a *Table*.

The real data resides in the `worksheets` directory. Here, every file contains the data of one of the *Worksheets* that is part of the *Workbook*. Within this file lies a two-dimensional representation of the values and formulae of the Excel worksheet. Transforming these files is done by looping over all `<c>` elements in the XML file and converting them to *Cells* in the *Spreadsheet*. The type of the cell is based on the existence of the `<f>` and `<v>` elements:

- if both are present, we consider the cell to be a *Formula Cell*;
- if only the `<v>` element is present, we consider the cell to be a *Value Cell*;
- if both are omitted, it is an *Empty Cell* and it will not be stored in the *Spreadsheet*.

Like we discussed in Section 2.2.2, a formula for a *Formula Cell* needs to be parsed to their own DSL. We use the work of E. Aivaloglou *et al.* [44]: their `XLParser` tool. This tool has a 99.9% successful parse

```

<worksheet>
  <sheetPr codeName="Sheet1">
    <tabColor theme="9" />
  </sheetPr>
  <dimension ref="A1:M25" />
  <cols>
    ...
  </cols>
  <sheetData>
    ...
    <row r="17" spans="1:13" ht="30" customHeight="1" x14ac:dyDescent="0.45">
      <c r="A17" s="10" />
      <c r="B17" s="14" />
      <c r="C17" s="38" t="s">
        <v>52</v>
      </c>
      <c r="D17" s="26">
        <v>2500</v>
      </c>
      <c r="E17" s="26">
        <v>1500</v>
      </c>
      <c r="F17" s="27">
        <f>E17-D17</f>
        <v>-1000</v>
      </c>
      <c r="G17" s="14" />
      <c r="H17" s="14" />
      <c r="I17" s="14" />
      <c r="J17" s="14" />
      <c r="K17" s="14" />
      <c r="L17" s="14" />
    </row>
    <row r="18" spans="1:13" ht="25" customHeight="1" x14ac:dyDescent="0.45">
      <c r="A18" s="10" />
      <c r="B18" s="14" />
      <c r="C18" s="38" />
      <c r="D18" s="26" />
      <c r="E18" s="26" />
      <c r="F18" s="27" />
      <c r="G18" s="14" />
      <c r="H18" s="14" />
      <c r="I18" s="14" />
      <c r="J18" s="14" />
      <c r="K18" s="14" />
      <c r="L18" s="14" />
    </row>
    ...
  </sheetData>
</worksheet>

```

Listing 2.6: An example of an XML file in the `worksheets` directory of Listing 2.5. This file stores the contents of the worksheet in `sheetData` as a matrix. The file has been altered for legibility, removing XML metadata and information about markup in the worksheet.

rate and produces parse trees of the formula [44]. This parse tree is converted to the *Formula Model* and is used to construct the *Formula Cell*.

These two parsers create the *Formula Model* and the *Structural Model*. Since the *Formula Model* is part of the *Structural Model*—it is a sub-intermediate representation—constructing the model is done in one compiler step. Furthermore, since we do not have any extra compiler steps in this phase of the compiler, we can hand off the representation of the Excel Sheet, the *Structural Model* to the next phase: the *Compute Phase*.

## 2.3. Deriving the Logic

Now that we have parsed the Excel file to an internal data structure called the *Structural Model* in the previous section, we can start to analyse the *Workbook* and begin deriving the logic hidden behind the cells. The real magic happens under the hood, where a network of formulas calculates the values of the formula cells. We want to extract this magic from the spreadsheet and translate it to actual code. This is what the *Compute Phase* does: it extracts the computational model found in the network of linked formulas and cells and creates a dependency graph.

Like we discussed in the high level overview in Section 2.1, the *Compute Phase* utilises the *Compute Model* to represent the underlying computational model as a *Compute Graph*. In this section, we cover the *Compute Model*, its relation to the *Compute Graph*, and a compiler step to enrich the *Compute Model* even further. First, we introduce the *Compute Unit*, and its place in the *Compute Model*. Then, we introduce the *Compute Graph* and how it is constructed from the *Structural Model*. Finally, we cover the type inference compiler step that enhances the model with type data needed for code compilation.

### 2.3.1. Compute Model

The *Compute Model* is a model of the underlying computational model. This intermediate representation provides a way to express computations and is used in different ways. In the ‘basic’ compiler, the *Compute Model* is ultimately represented as a graph of interconnected nodes that compute the value in the desired output cell. In the next chapter, we expand upon this by introducing another way to represent this model.

At the core of the *Compute Model* lies the *Compute Unit*. This unit represents a basic operation with input and output. *Compute Units* can be connected to each other, forming a network or flow of computations. When the result of *Compute Unit B* is used as input of *Compute Unit A*, we say that unit *B* is a *dependency* of unit *A*. Conversely, unit *A* is a *dependent* of unit *B*.

```
record ComputeUnit(Type type, Location Location, ComputeUnit[] Dependencies);
  → record Nil();
  → record ConstantValue(object Value);
  → record Input(string Name);
  → record Function(string Name);
```

Listing 2.7: The Compute IR

An overview of available *Compute Units* can be seen in Listing 2.7. *Compute Units* have a few properties in common:

- A list of dependencies. This represents the *Compute Units* this unit depends on. For some elements like the *Constant Value*, this list will always be empty since they do not depend on any value.
- A location in the spreadsheet. This location is used for traceability within the compiler so we know where this computation would have taken place in the spreadsheet. When extending the compiler, this can be of great help while debugging. Furthermore, we will use this location as a heuristic for a better code structure in the next section when we compile the *Compute Model* to the *Code Model*.
- Every *Compute Unit* has a type. This type represent the type of the value that the *Compute Unit* outputs.

The *Compute Unit*, like an Excel Formula, represents a way of computing a value. As we foreshadowed in Section 2.2.2, the *Compute Model* has a lot in common with the *Formula Model* as it models the Excel Formula Language as well. The *Compute Model* closely resembles the *Formula Expression*. However, there are a few differences. The first difference is the way the two models are constructed: the *Formula Model* is constructed like a tree. In the *Compute Model*, graphs will form if there are multiple dependencies on one Compute Unit. Furthermore, the next chapter will extend the *Compute Model*,

introducing new *Compute Units* and essentially making the *Compute Model* a superset of the *Formula Model*.

The individual *Compute Units* that construct the *Compute Model* can be directly derived from their *Formula Model* counterpart. We discuss the similarities between the *Compute Units* and *Formula Expressions* since we cover the transformation between these nodes when we transform the *Structural Model* to the *Compute Graph* in the next section.

**Constant Value.** A *Constant Value* is a value of any type. The *Constant* stores the data like a string ( 'String' ) or numeric value ( 42 ) and directly relates to the constant in the *Formula Model*. Often, the constant is the dependent of other *Compute Units*. However, it can also be an actual constant used within a formula, like the 2 in  $=2 * F3$  , or a value that is extracted from a *Value Cell*. For instance, if F3 in the previous example is a *Value Cell*, then its contents will be converted to a *Constant*.

**Function.** The *Function* is a *Compute Unit* that computes data from their dependencies, which are essentially the *arguments* for the *Function*. The function is stored as its name and the arguments.

**Nil.** *Nil* is a computation that computes nothing. We use *Nil* to denote empty cells often found in ranges. In subsequent compiler steps, *Nil* is often converted to a default value or removed altogether. This is a new addition to the *Compute Model* and is not represented by the *Formula Model*.

**Input.** Up until this point, we have not considered the input of the user yet. Within the *Compute Phase*, we introduce the *Input Compute Unit*  $In\{N\}$  , which represents a value that is not known at compile time. The *Input* will always take the type of the cell it replaces.

We do not have any references in the *Compute Model*. This distinction from the *Formula Model* comes from the fact that the *Compute Model* links the dependencies to other cells. When all dependencies are removed, we get a web of interconnected *Compute Units*. We call this web the *Compute Graph*.

## 2.3.2. Compute Graph

The next step in compilation is the conversion to the *Compute Graph*, linking the compute units in the individual cells to each-other, creating one big graph. The compute graph is a uni-directional non-cyclic graph that describes the underlying compute model of an Excel sheet. The *Compute Graph* can be also seen as a *dependency graph*. The compute graph used in this thesis resembles the support graphs found in Excel or the open implementation by P. Sestoft [8], which also store computations as nodes and use dependencies to create a dependency graph. The *Compute Graph* also resembles a *Value Dependency Graph* [45], as we only store the data flow of the Excel calculations.

Within the 'basic' compiler, the *Compute Graph* fully represents the *Compute Model*. Unlike the previous phase, there are two compiler steps that enhance the *Compute Graph* and therefore the *Compute Model*. In this subsection, we cover the conversion of the *Structure Model* to the *Compute Graph*. Then, we discuss the two compiler steps: input insertion and type inference.

### Model to Model

For the conversion of the *Structure Model* to the *Compute Graph* we look at the outputs the user has specified and convert that cell from the *Structure Model* to a tree of *Compute Units*. This is recursive process, since the references of cell, and their references, etc, will be converted too. When a cell has been fully converted, it will be linked to its parent: it will become a dependency of the parent.

To be more precise, for every cell that we convert from the *Structure Model* we distinguish between the type of the cell:

1. The cell is a *Value Cell*: the value and type of the cell is copied to a *Constant Value* compute unit.

2. The cell is a *Formula Cell*: The formula is transformed from the *Formula Model* to the *Compute Model*: Every *function* and *operator* is converted to a *Function Compute Unit* and constant values are converted to *Constant Compute Unit*. When we encounter a *Reference*, we convert the cells in the Reference using the same method, recursively. When the cells are converted, they are added as a dependency to the dependent of the reference.

When a cell is converted, it is added to a dictionary that is checked before converting a new cell. When the cell that we want to convert is already converted to a *Compute Unit*, we use that *Compute Unit*. Else, we convert it using the above algorithm.

Take the *Structure Model* of a simple spreadsheet in Spreadsheet 2.2 for example. If B3 is considered an output, this is where the conversion starts. It tries to convert the *Formula Cell* and converts it to `Function('SUM', [])`. It contains a *Range Dependency* and as such, cells B1 and B2 also have to be converted. We convert B1 (`=2*A1`) to `Function('*', [2, 42])` after converting A1 to the *Constant Value* 42. If we want to convert B2 (`=1/B1 + A2`) we need to convert B1 again. However, it is present in the dictionary, so we use the already converted B1. B2 is converted to `Function('+', [Function('/', [1, Function('*', [2, 42])]), 1000])`. Both B1 and B2 are then added as dependency to B3.

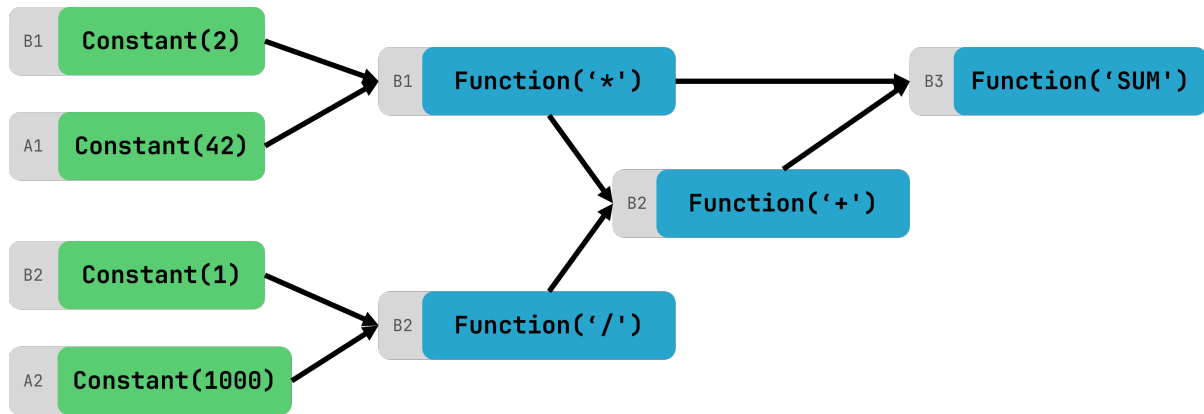


Figure 2.3: The fully constructed compute graph of Spreadsheet 2.2. It is a graph due to the dependencies of B3 on B1 and B2, and the dependency of B2 on B1.

An important side effect of this is that we only use the cells that are used in the calculation of the user specified output and ignore the rest. This makes this compilation step also an optimisation since we remove any cells that would be *dead code*.

## Traversing the Compute Graph

The compute graph supports the operation for traversing the graph. This is done in topologically sorted way, which means that when traversing node *a*, we have already traversed the dependencies of *a*. This ensures consistent traversal and updating of the graph. Traversing the graph and making updates to the graph is a common operation within a compiler step.

## Input Substitution

	A	B
1	42	=2*A1
2	1.000	=1/B1 + A2
3	600	=SUM(B1:B2)

Spreadsheet 2.2: A simple representation of the *Structure Model* of a spreadsheet that contains several cells that are referenced by formulae in other cells.

The compiler takes cell references (*Locations* like 'Sheet1'!A1 ) as input that can be used as parameters for the generated code. These inputs need to be inserted into the graph. Utilising the graph traversal algorithm discussed in the previous section, we find the first Compute Unit with a location of one of the inputs. This *Compute Unit* is replaced with an *Input Compute Unit*. Since the input compute unit represents a future value, it does not have any dependencies. As such, the dependencies of the *Compute Unit* it replaced will be pruned unless they are referenced by other *Compute Units* in the graph.

## Type Resolution

In general, when compiling to strongly-typed languages, you need to know the types of the computations you want to model. We already saw in Section 2.2 that Excel provides the types at cell level: we used those to infer the types of the cells in a structure and could infer the type of a column based on that. The types we discovered in Section 2.2 do not cover the formulas. These types are automatically inferred by Excel and are not disclosed when parsing them. In other words, we need to resolve the types for individual compute units.

Since we do know the types of the leaves of the graph, as they are constant values and have been given a type from the start, we can infer the types of their dependents, and thus recursively build a typed Compute Graph. In order to know what the type is of a certain compute unit, we rely on the types of the dependencies and an inference rule. This inference rule describes what a valid inference is for this compute unit, and what its type might be if there is a valid inference. There can be multiple inference rules for one compute unit.

For instance, take the *Function* compute unit, especially the  $F(' +')$  compute unit, which is basic addition between two dependencies. See the inference rule in Rule 2.1, which uses some syntactic sugar to state that the function  $F(' +')$  can be type  $\tau$  if and only if it has two arguments: which is denoted by the pattern matching  $[a1, a2]$  that denotes that the list should have two elements  $a1$  and  $a2$ ; and that those two arguments are both of the same type.

$$\frac{a1 : \tau \quad a2 : \tau}{F(' - ', [a1, a2]) : \tau}$$

Rule 2.1: The inference rule for the  $F(' -')$  compute unit, stating that it needs 2 arguments, and both arguments need to be of the same type.

Some functions, such as  $RAND()$ , return a random number and does not take any inputs. These functions will always be one type. Rule 2.2 shows the inference rule for such a function. The same is true for *Constant Value* compute units. These already have their types determined from the *Worksheet* or from the parse tree of a *Formula*, as we discussed in previous sections.

$$\frac{\top}{F(' RAND ', []) : \text{int}}$$

Rule 2.2: The inference rule for the  $F(' RAND')$  compute unit. It is always an integer.

**Limitations.** The compiler currently lacks support for Excel's IF function due to the thesis scope, but it is worth noting a limitation in the existing type-inference rules. In Excel, an if-statement is a normal function  $IF(t1, t2, t3)$  and thus maps to  $F("IF", [t1, t2, t3])$  in the *Compute Model*. A naive typing rule like Rule 2.3 would require the two branch expression to share the same type, so that the outcome of the IF function also has this type.

$$\frac{d : \text{bool} \quad b1 : \tau \quad b2 : \tau}{F(' IF ', [d, b1, b2]) : \tau}$$

Rule 2.3: The inference rule for the  $F(' RAND')$  compute unit. It is always an integer.

A	
1	TRUE
2	=IF(A1, 100, TRUE)
3	=IF(A1, ISNUMBER(A2), A2)

Spreadsheet 2.3: A simple representation of the *Structure Model* of a spreadsheet that contains several cells that are referenced by formulae in other cells.

However, Excel allows these two branch expressions to return different types. The proposed inference system is unable to represent this, as the compute unit containing the if-statement would have more than one possible type. This calls for a type system with union types or type refinements based on the condition of the if-statement.

## 2.4. Generating Code

When we have finished transforming and optimising the *Compute Model* and have created a representation of the computational model, it is finally time to produce actual code. The final phase of the compiler, the *Code Phase* deals with this problem. During the code phase, the compute graph is converted to the code layout model, and then converted to actual programming languages. In this thesis, we transform the code to C# code.

The *Code Phase* has two parts: (1) the language-agnostic part, where we convert the *Compute Model* into the language-agnostic *Code Model*; (2) the language-specific part, where we compile the *Code Model* into C# code. Most optimisations done in this phase are done on the language-agnostic *Code Model*. The *Code Model* is a model that represents an abstraction of code. The *Code Model* is built for easy manipulation and supports both object-oriented and functional features. Consequently, the *Code Model* is able to model a lot of programming languages. The use of the *Code Model* prepares the compiler for compilation to other languages than C#, such as Java.

In this section, we will first introduce the *Code Model* in more detail. We discuss the use of the *Let* node to simplify manipulation and present further compiler steps possible for the language-agnostic *Code Model*. Then, we discuss the final part of compilation: the transformation from the *Code Model* to C#.

### 2.4.1. Code Model

The *Code Model* is the intermediate representation that provides structural guidance in emitting correct code and ease final transformation of the code. The model is a simplified abstract syntax tree and simplifies many parts of a normal parse tree and omits implied syntax.

In comparison with the Roslyn compiler model (introduced in Section 2.4.2) the *Code Model* is much simpler. The Roslyn compiler model is used to model the AST of .NET languages. It allows for complete modification of the syntax of the language. However, this can also result in invalid C# syntax. For instance, C# has a `var` type that automatically infers the type at compile time, like the `auto` type in C++. This ‘type’ can only be used as a direct or simple type, not as part of a complex or generic type, i.e. `var i = 0;` is valid code, but `List<var> = [1,2,3];` is not. With the Roslyn Compiler, it is possible to create such a type. Instead, the *Code Model* strictly forbids this.

Listing 2.8 presents an overview of the *Code Model*. Most of the nodes of this IR can be recognised from popular object-oriented and procedural programming languages, such as *Classes*, *Methods* and *Function Calls*. As such, many elements will not be discussed. That said, some elements of the model, such as the *Let* expression, require more elaboration.

```

Type(String name)
-> Class(String name, Property[] members, Method[] methods)
-> ListOf(Type member)

Method(String name, Statement[] body)

Property(String name, Type type, Expression? init, Expression? get, Expression? set);

Statement()
-> If(Expression cond, Statement[] then, Statement[] else)
-> ExpressionStatement(Expression expression)
-> Declaration(Variable variable, Expression value)
-> Return(Expression expression);

Expression(Type type)
-> Assignment(Variable variable, Expression value)
-> Constant(Type type, Object Value)
-> FunctionCall(Expression? self, string name, Expression[] arguments)
-> Lambda(Variable[] parameters, Expression body)
-> Let(Assignment variable, Expression in)
-> List()
-> ListAccessor(Expression list, Expression index)
-> MapAccessor(Expression map, Expression index)
-> Property(Expression self, String name)
-> Variable(String name)

```

Listing 2.8: A formal definition of the *Code Model* IR.

## Statements

The IR is able to model standard statements like variable declarations, return and if-statements. Those are synonymous with the statements in many popular languages. That said, an interesting node of the model is the *Expression Statement*. This statement models an expression that is used as a statement. Usually, this is a function call or an assignment. For example, in the C# language, a call to write something to the console is done through the `Console.WriteLine()` function call. In the *Code Model*, this would be modelled as an *Expression Statement* containing a *Function Call* expression.

## Let

Many of the expressions are straightforward and are directly copied from popular object-oriented and procedural programming languages. However, transforming object-oriented code can have its hassles. If we want to transform the body of a method with procedural statements, it can become quite tricky when working with both expressions and statements. For instance, let's say we want to transform the following snippet of code:

```

...
double average = [1, 2, 3, 4, 5].Sum() / 5d;
...

to

...
int[] array = [1,2,3,4,5];
double average = array.Sum() / 5d;
...

```

Doing this conceptually may not seem hard: we just extract the array to a new variable and replace the array with the new variable. However, doing this with a generic algorithm requires the algorithm to keep track of where we are in the method body as we need to insert a statement in the list, and requires a separate traversal of the expression to replace the list. This example signifies that this seemingly simple refactor step requires a complex algorithm.

As such, the *Code Model* introduces a known concept in functional programming languages: the *Let* expression. This expression essentially models the assignment statement, but does this in an expression. As such, the statements from above can be expressed in one expression:

```

let average = (
  let array = [1,2,3,4,5]
  in array.Sum() / 5d;
)
in ...

```

This simplifies the algorithm described above, as we only need to concern with expressions. Instead of keeping track of where we are, we only need a single traversal of the expression. We insert a *Let* statement around the expression and replace the list with the variable.

## Types

*Types* are an important notion in the *Code Model*. Many high-level programming languages are strictly typed languages, which means that everything should have a type at compile time. As such, our model requires the typing of every single node in the syntax tree.

Many types are automatically converted when we convert the compute graph to the code layout model. Furthermore, many expressions and statements do not require their own types, but can infer their types from the types of their children. For instance, take the `ListExpression` which is always of the `ListOf` complex type. The `ListExpression` models a sequence of values of the same type  $\tau$ , and as such, the `ListExpression` will always be of type `ListOf( $\tau$ )`

**Precision.** As we discussed in Section 2.2, Excel does not differentiate between integers and doubles. Under the hood, Excel follows the IEEE 754 Floating-Point Arithmetic specification [46]. It utilises a double precision floating point value for the implementation of ‘numeric values’ [46]. This results in 15 digits of significant precision [47]. Within this Excel Compiler, we also use double precision floating point values for representing the numeric values.

Precision is key, especially in the actuarial calculations context. We do not want to lose a few decimals due to precision issues, which would propagate and mean that the pension fund would be paying more or—even worse—less than what you should have gotten.

### 2.4.2. Creating the model

Having covered the most important elements of the *Code Model*, we can discuss how to create it from the *Compute Model* and convert it to code. This is done in four steps. First (1), we transform the *Compute Model* into the *Code Model* by mapping the different nodes from the *Compute Model* to *Expressions* in the *Code Model*. Simultaneously, we introduce variables by using the references to the *Structure Model* found in the *Compute Model*. Then we iterate upon that model and (2) inline variables and (3) transform the *Let* expressions into statements. Finally (4), we emit the code layout model as code to the disk. In the upcoming sections, we discuss these steps further.

## Variables

The first step of the code layout model is the introduction of variables. Like we covered in Section 2.3, the *Compute Model* is essentially one big expression that needs to be evaluated. However, emitting this big expression will significantly decrease the readability of the program. As such, we need to introduce variables to make sense of the big sub-expressions.

Every *Compute Unit* contains the location it originated from in the *Spreadsheet* of the *Structure Model*. In order to increase legibility (the alternative would be randomly generated values), traceability and resemblance to the original Excel file, we use these locations to store computations.

In more precise terms, we use a top-down approach, starting at a root of the *Compute Graph*, and transform every node to a *Code Model* node. Every time we encounter a *Compute Unit* that is from a

new cell, we begin constructing a new variable and put the other—already constructed one—in a *Let* expression. In other words, we recursively build a big expression of *Lets*.

## Inlining

A common practice in an Excel file is to display the values from another sheet in a separate cell, and then use that cell for calculations in the sheet. This essentially makes the display cell a proxy to the other sheets. As a result of the variable creation compiler step discussed in the previous section, this creates a variable that is being assigned to another variable:

```
double interestJ12 = interestF65 - interestF5 - interestJ11;  
double monthlyBudgetReportJ7 = interestJ12;  
double monthlyBudgetReportD10 = monthlyBudgetReportJ7;
```

In this optimisation pass, these proxy variables will be removed. We do this by refactoring the `Let` nodes, checking if a `Let` node contains another `Let` node with the assignment just a variable name. In the end, we will remove this node, and just assign the value directly to the last variable. As such, the example above becomes:

```
double monthlyBudgetReportD10 = interestF65 - interestF5 - interestJ11;
```

and is instantly more readable.

## Statements

After inlining, we begin the language-specific part, as we need to convert the *Let* expression to statements, since C# does not support the *Let* expression. As such, we employ a straightforward algorithm that converts every assignment in every *Let* expression to a statement. The following code layout model would be transformed:

```
DeclarationStatement z (Let y = 10  
                        in let x = 10  
                        in x + y)
```

into:

```
DeclarationStatement y 10  
DeclarationStatement x 10  
DeclarationStatement z (x + y)
```

The current layout model is looking more and more like a high level programming language. This example also exemplifies the versatility of the code layout model, as we can have two different styles for variable declaration that are highly coupled in parallel. This versatility is crucial when supporting different programming languages, and makes the EXCELERATE compiler ready for future expansion.

## Emission

Just before emission, at the end of the *Code Phase*, we have constructed a language-agnostic representation of the Excel computation. The final challenge is to convert this abstract model into concrete compilable code. In this thesis, we chose C# for the target or destination language. In this subsection, we briefly cover the Roslyn API to explain what it does. Then we dive into the code generation and explain how we map the code: a straightforward process. Finally, we discuss some of the subtleties needed to make the project actually compile.

**Target Language.** We choose C# for several reasons. C# is a multi-paradigm language, which means it is possible to use both object-oriented imperative or functional code. Another reason is that C# is one of the most widely used languages, with the .NET framework being the most used framework [48]. Lastly, the author has the most experience with the C# language and we believe this is one of

the most readable languages, but we acknowledge this is purely preference. That said, the language-agnostic nature of the *Code Model* means that the compiler can be extended to Java, Kotlin or another programming language

**Roslyn API.** The Roslyn API is an open-source .NET compiler platform developed by Microsoft. It exposes the whole compiler process, from internal data structures to transformations, to the programmer and let's the programmer use the compiler within the C# language itself.

Like we spoke about earlier, the Roslyn API is flexible and expressive. However, this flexibility comes at a price since it demands explicit syntax and is able to produce invalid C# code when misconfigured. While there are helper APIs to combat this issue, they are still very primitive, verbose, and allow for uncompileable code. Hence, we rely on the stricter semantics the *Code Model* grants.

It is important to emphasise that this Roslyn step, by design, does not improve the code generated by the layout model. We only define what it has to output and it renders this to a source file. This places the burden of correctness on the layout model and the transformations leading up to emission. This design choice ensures transparency and reusability: every optimisation, transformation, or refactoring is explicit in the layout or preceding passes, and not hidden inside the emission step for a specific language. This step does, however, apply some language-specific transformations to improve readability. These optimisations include choosing for one-liner if-statements instead of full bodied if-statements or making sure the latest language features are being used. It does this on the Roslyn syntax tree as it is being generated.

**Mapping.** Translating from the *Code Model* to the Roslyn syntax tree is fairly straightforward. Statements and expressions are recursively mapped to Roslyn's corresponding syntax nodes. For many constructs, this direct, and not much work is needed. For example, a list initialisation in the layout model becomes an object creation with a collection initialiser in C#. For others, such as properties and functions, we need to see if optimisations can be made to the code layout, such as using onliners with expression bodies versus full scoped bodies.

Furthermore, we rely on a few helper methods from the *Code Model* to create constructors for types, as they are not explicitly defined in the model but rather derived from the settable data in the structures.

**Layout.** During emission, we must also address the organisation of code into files, namespaces, and classes. The code layout model describes a project as a collection of classes and methods, but leaves file system organisation abstract. The emission pass generates a dedicated C# file for each top-level class, placing them within a common namespace so they can be accessed easily. This is a common practice in C# [39].

## 2.5. Reflecting on the Compiler

The three phases of the compiler: *Structure*, *Compute*, and *Code* produce a class library with the same semantics as the Excel file. In this section, we discuss the readability of the code, reflecting on the current compiler, and highlighting areas that can be improved.

### 2.5.1. Two Examples

Before we begin the discussion, we present two examples of Excel files that have been compiled to C#. These files are small so the code fits on the page. The first spreadsheet is the same as Spreadsheet 2.1 in the High Level Overview in Section 2.1. It describes a calculation of the Difference between Projected and Actual income. This example has many independent code paths, such as the calculation of the differences: to calculate the difference of one row, we do not have dependencies on the other row.

	A	B	C	D
1		<b>Projected</b>	<b>Actual</b>	<b>Difference</b>
2	Income 1	6.000	5.800	=C2-B2
3	Income 2	1.000	2.300	=C3-B3
4	Extra Income	2.500	1.500	=C4-B4
5	TOTAL			=SUM(D2:D4)

Spreadsheet 2.4: An extract of the 'Family monthly budget' sheet in the 'Family monthly budget'. This spreadsheet calculates the differences in projected and actual income.

```
public class Program
{
    public double Main()
    {
        double sheet1D2 = 5800 - 6000;
        double sheet1D3 = 2300 - 1000;
        double sheet1D4 = 1500 - 2500;
        double sheet1D5 = new List<double> { sheet1D2, sheet1D3, sheet1D4 }.Sum();
        return sheet1D5;
    }
}
```

Listing 2.9: A compiled version of the spreadsheet in Spreadsheet 2.4 using the basic compiler described in Chapter 2.

Listing 2.9 shows the compiled code if we run the compiler with `D5` specified as output cell. The *Inline Variables* compiler step has inlined the constants into the difference calculations.

The second spreadsheet contains a linear calculation path, calculating the interest and current balance in a savings account. It can be seen in Spreadsheet 2.5. This spreadsheet was compiled using the 'basic' compiler with `C5` as output cell configured. The code can be found in Listing 2.10.

	A	B	C	D
1		<b>Interest</b>	<b>Balance</b>	<b>Deposit</b>
2	May		10 000	
3	June	=0.01 · C2	=C2+B3+D3	500
4	July	=0.01 · C3	=C3+B4+D4	500
5	August	=0.01 · C4	=C4+B5+D5	500

Spreadsheet 2.5: An extract of the 'Interest' sheet in the 'Family monthly budget' Excel file. This spreadsheet calculates the

```
public class Program
{
    public double Main()
    {
        double sheet1C2 = 10000;
        double sheet1B3 = 0.01 * sheet1C2;
        double sheet1C3 = sheet1C2 + sheet1B3 + 500;
        double sheet1B4 = 0.01 * sheet1C3;
        double sheet1C4 = sheet1C3 + sheet1B4 + 500;
        double sheet1B5 = 0.01 * sheet1C4;
        double sheet1C5 = sheet1C4 + sheet1B5 + 500;
        return sheet1C5;
    }
}
```

Listing 2.10: A compiled version of Spreadsheet 2.5 using the basic compiler described in Chapter 2. The code shows a repetition in calculating the cells in the *Interest* and *Balance* column.

## 2.5.2. Missing structure

When evaluating the compiled code, two things are investigated: the semantics and the readability and extensibility. The compiler should emit code that is semantically equivalent to Excel—without this, the compiled code would require extra human intervention to be useful. Running the evaluation as described in Section 4.1, it produces the same output as Excel. As such, we can assume the semantics of Excel are preserved.

As discussed previously in Section 1.2.4, the code the compiler emits should be readable and extensible: the code written should be on the same level as how a good software engineer would have written the code. Applying this arguably subjective definition on the two samples, we argue a few problems emerge.

Listing 2.9 is considered good, simple code without duplication and contains use of idiomatic LINQ constructs (SUM) rather than using loops. This reduces the cognitive complexity and increases comprehensibility and readability. On the other hand, the code in Listing 2.10 is less readable. It contains lots of duplicate code, since the calculation for the balance is repeated three times. Furthermore, since the values in the *Deposit* column are inlined, it is not clear that these are variable values and look like one single constant value added to the balance.

The compiled versions are currently without parameters. When we add parameters to Listing 2.9, we quickly discover a problem: there will be too many parameters. For every row, we would have two parameters, resulting in six parameters for this simple problem. Expanding the spreadsheet would result in even more parameters. This could be fixed with an *Input* class containing all inputs as properties, but that is not idiomatic.

Another clear issue in both listings is the lack of extensibility. The Excel spreadsheets may be expanded to include the month *September* in the savings calculation, or remove a source of income in the *family budget*. In both cases, the compiled code would have to be recompiled. Ideally, the compiled code is flexible that it should not have to be recompiled, since the underlying computational model does not change: the only change is the extra argument in the SUM and how that argument is calculated, but that is done in the same way as the other rows.

## 2.5.3. Guided by structure

A way to solve the problems is to analyse the code and introduce refactoring. For instance, the repetition in Listing 2.10 could be fixed by statically analysing the code, discovering this recurrence of calculations and abstracting the repetition. However, for the problem of extensibility and parameters,

```
public record Income(double Projected, double Actual) {
    public double Difference => Actual - Projected;
}

public class Program
{
    public double Main()
    {
        List<Income> incomes = [
            new(6000, 5800),
            new(1000, 2300),
            new(2500, 1500)
        ]

        double differences = incomes.Select(i => i.Difference).Sum();
        return differences;
    }
}
```

Listing 2.11: A more idiomatic version of Spreadsheet 2.4. Written by the author.

the fix is not easy. A solution would be to introduce a structure that models the data in such a way that it fits the computations, like in Listing 2.11. This introduces a new class `Income` that models the rows found in the spreadsheet. It also creates a computed property `Difference` that would have removed duplicated code. Furthermore, it uses the `Select` LINQ statement to only sum the differences. Furthermore, if we take this new `incomes` local variable as parameter instead, adding or removing rows from the income is easier than ever without creating any more parameters—increasing the extensibility at no expense of readability.

However, creating these structures with the *Code Model* alone will be hard: detecting which operations should be part of the structure is hard since the scope of the structure is highly dependent on the context. Besides, in small examples like Spreadsheet 2.4 and Spreadsheet 2.5 this might be do-able but uncovering possibly multiple structures within a large spreadsheet with many data is highly complex.

That said, a similarity can be found between the newly added `Income` structure and the *structure* of the *Spreadsheet* in Spreadsheet 2.4. If we see this spreadsheet as a table, with columns *Projected*, *Actual*, and *Difference*, the structure basically describes one row of this table. This demonstrates the fact that the semantics is often encoded in the spreadsheet structure. The meaning of the nodes used in the *Code Model* can be inferred from the structure. For instance, The `SUM` in `D5` does not just denote the sum of the range, but more likely means the total of the *difference* column of the table.

When we use this heuristic as part of the compiler, we are able to create more readable code that fits more closely to the semantics of the spreadsheet. In the next chapter, we introduce exactly this form of compilation, called *structure-aware compilation*, that takes structural information and metadata into account in order to improve the readability of the emitted code.

*Chapter 3*

**EXCELERATE**

3.1. Structure-aware  
    compilation ..... 35

3.2. Changes to the  
    Compiler ..... 38

3.3. Finding the Structures ... 40

3.4. Embedding the  
    Structures ..... 45

3.5. Coding the Structures ... 49

3.6. Discussion of the Compiler  
    Design ..... 52



The ‘basic’ compiler introduced in the previous chapter had difficulties emitting idiomatic code. However, as we discussed in Section 2.5, the desired idiomatic code had a lot of structural similarities with the spreadsheet it should represent. We found that these structures could help compilation and improve the readability of the code. In this chapter, we introduce EXCELERATE and its *structure-aware compilation*. This technique uses structures embedded in the spreadsheet as heuristics during compilation.

At first glance, an Excel workbook might seem like a collection of mere numbers and formulas, arranged in rows and columns. Yet, beneath this seemingly simple grid lies a carefully orchestrated structure that not only conveys data but also encodes semantics and context.

Consider the family budget spreadsheet introduced in the introduction in Figure 1.1. The data and formulas alone do not communicate the semantics of the spreadsheet well. The styles and structures used in the spreadsheet is what transforms these pieces of data into an understandable presentation. As such, the challenge lies in preserving the readability and implicit structure that gives the spreadsheet its meaning.

In this chapter, we start by introducing *structure-aware compilation* and covering the structures we identified. Then, we proceed to list the changes that need to be made to the compiler by giving a high level overview of what needs to happen to compile using *structure-aware compilation*. The rest of the chapter is dedicated to the specifics of changes needed in each phase.

### 3.1. Structure-aware compilation

When the ‘basic’ compiler converted the *Structure Model* to the *Compute Graph*, a lot of metadata about the spreadsheet was lost. This metadata—in the form of the 2D formation on the *Spreadsheet* grid—contains valuable data that could be used as heuristics in compilation.

In this thesis, we introduce *structure-aware compilation*: a compilation technique that uses knowledge of the structure of the source program to guide code generation to the target language. In essence, every source program should be able to compile to the target language, but *structure-aware compilation* tries to preserve the structure of the source program in the emitted code.

For instance, take Spreadsheet 3.1. The left spreadsheet obviously contains a table. The right spreadsheet is computationally equivalent but structurally totally different. The underlying computational model is the same. However, using structure-aware compilation, the former spreadsheet should be compiled with more idiomatic code than the latter, because the compiler will be guided by the structure of the table.

That said, in order for *structure-aware compilation* to work, we need to identify specific structures that could help structure the emitted code. Within the grid of the *Spreadsheet*, we should find patterns of structures.

	A	B	C	D	
1		Projected	Actual	Difference	
2	Income 1	6.000	5.800	=C2-B2	
3	Income 2	1.000	2.300	=C3-B3	
4	Extra Income	2.500	1.500	=C4-B4	
5	TOTAL			=SUM(D2:D4)	

	A	B	C	D	E
1	6.000	1.000	2.500		=A1-A2
2	5.800	2.300	1.500		=B1-B2
3					=C1-C2
4	TOTAL	=SUM(E1:E3)			

Spreadsheet 3.1: Two computationally equivalent, but structurally different spreadsheets, if the left spreadsheet takes **D5** as output and the right takes **B4** as output. Both cells compute the exact same value, through the same computations.

### 3.1.1. Types of Structures

Using the Microsoft Create Excel Template repository, two spreadsheets were identified. The Microsoft Create Excel Template repository contains spreadsheets that are well-formatted, follow the “best-practices”, model spreadsheets for different industries, and often show patterns that users adopt in their own spreadsheets. The patterns that occurred the most were the *Table* like we saw in Spreadsheet 3.1 and the *Chain* that models a recurring operation like in Spreadsheet 2.5.

Another pattern was found in spreadsheets provided by DNB, which are more focussed on an actuarial context. Here, a lot of data of a single variable in a computation would reside in one spreadsheet. This pattern, which we call the *Data Pond*, was not implemented because of time.

	A	B	C	D	E
1	0.9324	0.2353	0.7263	0.1825	0.2182
2	0.5876	0.7531	0.2743	0.5404	0.0610
3	0.6234	0.7195	0.1423	0.2772	0.4714
4	0.3861	0.9621	0.8352	0.6922	0.9480
5	0.5937	0.5274	0.0450	0.5837	0.7037

Spreadsheet 3.1: A real world snippet of a *Data Pond* based on a spreadsheet published by DNB. This sheet, called ‘Renteparameter\_phi\_N’ represents a single variable in a larger equation. It is part of a workbook filled with data like this.

The next subsections describe the structures we encountered in the repository. These definitions are loose, and we will introduce the heuristics for detecting them in Section 3.3.3. In other words, what the compiler considers to be a *Chain* or *Table* is covered in the next section. What we describe in this section is the background of these structures, what separates them from each other and what they mean semantically.

### Tables

	A	B	C	D	E
1	<b>Expenses</b>				
2	<b>Description</b>	<b>Category</b>	<b>Projected Cost</b>	<b>Actual Cost</b>	<b>Difference</b>
3	Extracurricular activities	Children	40.00	40.00	=D2-C2
4	Sporting Events	Entertainment	0	40.00	=D3-C3
5	Fuel	Transportation	450	0	=D4-C4
6	Parking Fees	Transportation			=D5-C5
7	<b>TOTAL</b>		=SUM(C3:C6)	=SUM(D3:D6)	=SUM(E3:E6)

Spreadsheet 3.2: An example of a table structure found in a *Spreadsheet*. Cell A1 denotes the title, range A2:E2 describes the headers of the table. The actual data can be found in between the headers and the column in A3:E6. Finally, the footer is described in A7:E7.

Tables are essential in Excel. Almost all *Spreadsheets* contain some sort of Table. This is not surprising given the inherent tabular presentation of the Excel grid. Furthermore, *Tables* are an explicit feature of Excel, as we saw in Section 1.3.

Semantically, a table represents a list of objects. These objects are described by the data in the rows. For instance, in the table in Spreadsheet 3.2, the table describes a list of expenses. The properties of the objects are determined by the column names. In the aforementioned table, this means that the

properties of an expense are a description, a category, the projected cost, the actual cost and the difference.

The table pattern that was often observed was a rectangular region in the *Spreadsheet* containing data, a header and sometimes a title or a footer. The columns of the table may contain data, or they contain a formula. If a column has the same formula in every row, that column is a *Computed Column*.

The data in the table does not have to be constant, it may be expressed through a formula reference expression. For instance, instead of having a constant 40.00 for the projected cost for the Extracurricular activities, we could instead have a reference to another cell x1 in the spreadsheet that would calculate this cost (but is not shown here).

	A	B	C	D	E
1	<b>Expenses</b>				
2	<b>Description</b>	<b>Category</b>	<b>Projected Cost</b>	<b>Actual Cost</b>	<b>Difference</b>
3	Extracurricular activities	Children	40.00	40.00	=C2-D2
4	Sporting Events	Entertainment	0	40.00	=C3-D3
5	Fuel	Transportation	450	0	=C4-D4
6	Parking Fees	Transportation			=C5-D5
7	<b>TOTAL</b>		=SUM(C2:C5)	=SUM(D2:D5)	=SUM(E2:E5)

Spreadsheet 3.3: The dependencies for the rows in Spreadsheet 3.2. Cells only depend on cells in their rows.

An important property of the *Table* is that there are no dependencies between columns. Columns in a *Table* are independent of each other (apart from the footer, which may reference all the columns). Spreadsheet 3.3 shows the dependencies in Spreadsheet 3.2. All these dependencies are horizontal: the cells only depend on other cells in their respective rows. As such, Spreadsheet 3.4 is not a table, since C5 and B5 contain references to C4, which is not allowed in a table. Instead, we have identified another structure that support this.

### Chains

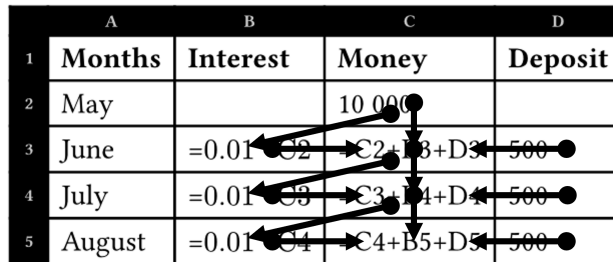
Chains are similar to tables, but they allow for dependent rows. Unlike *Tables*, chains are not a feature of Excel but are still common in Excel sheets since they describe a recursive calculation. Semantically, a chain represents a collection of objects that are in relation to eachother.

Take Spreadsheet 3.4, it represents a collection of balances for every month. Every new balance calculation is dependent on the previous balance. An object in a chain can just be constant data, such as the *Months* or *Deposit* column, or be calculated from the previous row.

This calculation is a recursive definition. As Spreadsheet 3.5 shows, the previous row is need to calculate a cell in the *Money* or *Interest* column. As with every recursive definition, a chain also has base cases, represented in C2 for Spreadsheet 3.4. Columns that use these recursive definition are called *Recursive Columns*.

	A	B	C	D
1	<b>Months</b>	<b>Interest</b>	<b>Money</b>	<b>Deposit</b>
2	May		10 000	
3	June	=0.01 · C2	=C2+B3+D3	500
4	July	=0.01 · C3	=C3+B4+D4	500
5	August	=0.01 · C4	=C4+B5+D5	500

Spreadsheet 3.4: An example of a chain-table. The chain is visible in the *Interest* (B) and *Money* (C) column of the table, where the interest calculated is based on the previous month.



Spreadsheet 3.5: The dependencies for the rows in Spreadsheet 3.2. Cells only depend on cells in their rows.

## 3.2. Changes to the Compiler

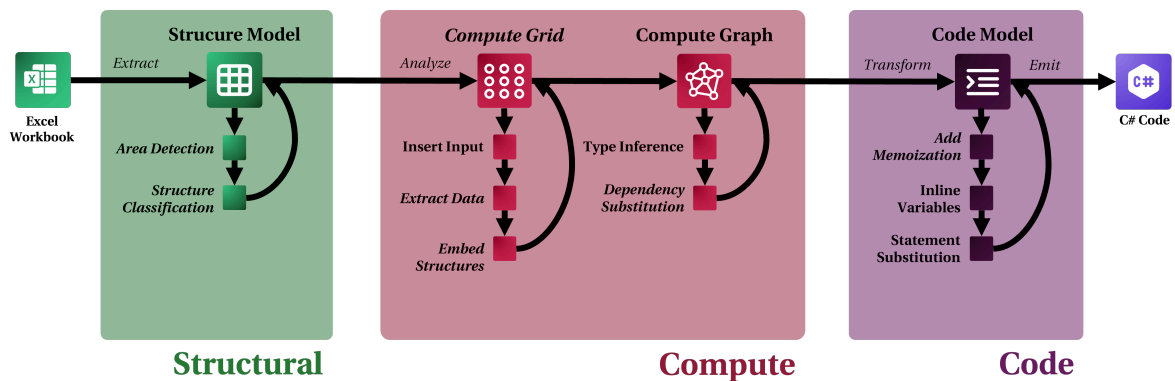


Figure 3.1: The new phases of the compiler that allow support for structures. The new models and compiler steps have been annotated by *cursive script*.

Now that we have an intuition for the structures found in the spreadsheet, we introduce EXCELERATE: an improved compiler which uses *structure-aware compilation* using the structures found in the previous section to emit more readable code. EXCELERATE builds upon the ‘basic’ compiler introduced in the previous chapter. It extends the *Structural* and *Compute Model* to allow for these new structures. In this section, we briefly discuss what is needed for the compiler to utilise these structures.

### 3.2.1. Changes in Input

A great side-effect of *structure-aware compilation* is that we can use these structures as input. The data residing in the structures does not have to be used. The programmer should be able to provide their own data. With the introduction of the *Chain* and the *Table*, we can exploit the structure to provide custom data by parametrising the structure, laying the burden of creating the actual structure on the developer working with the final code. In addition to cell inputs, EXCELERATE enables the programmer

to mark these structures as input. This enables flexibility and could improve readability as we saw in Section 2.5.

### 3.2.2. Detecting the structures

One of the immediate problems that arises is the detection in the spreadsheet. This detection should be done during the *Structural Phase* as it relates to the structure of the spreadsheet.

Ultimately, this problem can be decomposed into two subproblems:

1. Finding regions in the spreadsheet that may contain a structure.
2. Classifying these regions.

A closer look at the structures we found reveals a common property: the chain and the tables are both rectangular structures and are often densely populated with data or formulae. As such, potential regions should exploit this property. In Section 3.3.2 we introduce *Areas*, which exactly exhibit this property.

Then, once we have found potential regions, we should try to classify them. In Section 3.3.3, we discuss heuristics for both structures. As we will see, these heuristics are able to capture most structures, but there are some caveats. The found structures will be stored in the *Structural Model*.

### 3.2.3. Linking the structures

The *Structural Model* merely records that a structure exists, it intentionally only captures the spreadsheet and does not modify it. For instance, detecting Spreadsheet 3.2 does not change any references made to the cells in the table, they still reference the cells in the spreadsheet: `E3:E6` is still a range but actually represents the data in the `Difference` column of the table. As such, while structures are found within the spreadsheets, they are separate from the spreadsheet.

To express this, we should inject the structure into the *Compute Model* so that `E3:E6` does not represent a range but instead a column of the table. Doing this in the *Compute Graph* is not efficient. For every structure, we would need to traverse the whole graph to update the references made to the table.

Instead, we introduce the *Compute Grid*, a part of the *Compute Model* that mixes the grid-like structure of the *Structural Model* with *Compute Units* to represent the values and computations. Using the *Compute Grid*, we can inject special *Compute Units* describing the structures. For instance, the footer of the table will be updated to a *Table Footer Compute Unit* that describes the operation done on the column of the *Table*. When we transform the *Compute Grid* into the *Compute Graph*, we make sure that references like `E3:E6` actually reference the structure.

### 3.2.4. Converting to code

The references to the structures should also be reflected in the code. EXCELERATE does this by representing every structure as their own class. We need to add steps to the *Code Phase* to generate these structures and add constructors in the main function to create the structures to utilise them.

These changes should incorporate the structures into the compile flow and allow for more readable output. In the rest of this chapter, we describe what changed in the different models, and how we integrate the structures into the compiler to make more readable code.

## 3.3. Finding the Structures

The first step in integrating the structures is finding them. We already discussed a way to do this: we first find regions in the spreadsheet that may contain a structure and then try to classify these candidates. To facilitate this detection mechanism, we need to change the *Structural Model* to allow for *Structures* to be stored. In this section, we discuss these new additions to the *Structural Phase*.

### 3.3.1. Changes in the model

```
-- record Workbook(string Name, Table[] Tables, Spreadsheet[] Spreadsheets);
++ record Workbook(string Name, UserTable[] UserTables, Structures[] Structures, Spreadsheet[] Spreadsheets);

record Spreadsheet(string Name, Set<Cell> Cells);

record Cell(Location Location);
  → record EmptyCell();
  → record ValueCell<T>(T Value);
  → record FormulaCell(Formula Formula);

++ record Structure(string Id, Range Location)
++ → record Table(Column[] columns)
++ → record Chain(Column[] columns)
++
++ record Column(Range Location, string Name, string Type)

-- record Table(string Name, Range Location, string[] Columns);
++ record UserTable(string Name, Range Location, string[] Columns);
```

Listing 3.1: Changes in the formal *Structural Model*. The *Table* is renamed to *User Table* and the *Structures* are added.

The most notable change made in the model is the introduction of the *Structure*. This describes a structure in the spreadsheet. Every structure has a unique identifier and a location denoted by a *Range*. The identifier is derived from the title of the structure, or the location. For example, the identifier of Spreadsheet 3.2 would be *Expenses* since it has a title. On the other hand, Spreadsheet 3.4 does not have a title, and would get the `Sheet1A1D5` identifier.

As we have discussed, a *Structure* can be either a *Table* or a *Chain*. Both *Tables* and *Chains* have columns. These columns always have a location and a name. Based on the content, they are assigned a type. The possible types are *Data*, *Computed*, and *Recursive*. In Section 3.3.3, we cover what these types mean and when they are assigned.

The *Table* type was changed to *User Table* to distinguish it from the new *Table* structure. This used to represent an Excel Table (which is marked by the user) and is used in *Table References*. However, with the introduction of the more generic *Structures*, this node has become near obsolete. In EXCELERATE, this is purely used for the discovery of *Structures*: we try to convert the cells in the range of the *User Table* to a *Table* structure.

### 3.3.2. Areas

Before we can classify if a region is a table or a chain, we need to find this region first. This region in a *Spreadsheet* that could contain a *Structure* is called an *Area* within EXCELERATE.

#### Detecting Areas

Detecting these areas correctly is hard. Excel does not explicitly store these areas. What we perceive as a nicely bounded table, might be hard to detect due to ambiguity surrounding gaps or decorative formatting. Gaps can mean a blank separator between areas, or just a row or column with missing data. Furthermore, formatting can mean a separation between adjacent structures, or they signal a

	A	B	C	D	E	F	G	H
1	<b>Expenses</b>						<b>Table 2</b>	
2							<b>Col1</b>	<b>Col2</b>
3	<b>Description</b>	<b>Category</b>	<b>Projected</b>	<b>Actual</b>	<b>Difference</b>		890	130
4	Extra... activities	Children	40.00	40.00	=D2-C2		187	092
5					=D3-C3		<b>Table 3</b>	
6	School tuition	Children	100.00	100.00	=D4-C4		<b>Col X</b>	<b>Col Y</b>
7					=D5-C5		42	65
8	Concerts		50.00	40.00	=D6-C6		28	48
9							05	58
10	Movies	Entertainment	50.00	28.00	=D8-C8			
11	Music	Entertainment	50.00	30.00	=D9-C9			
12								

Spreadsheet 3.6: An example of a spreadsheet with gaps. Four areas will be detected using the adjacency heuristic: `A1:A1` , `A3:E8` , `G1:H9` and `A10:E11` . The gap in row 10 results in the split of the table. Table 2 and Table 3 don't have an empty separation row, resulting in one area instead of two. This highlight the difficulties of detecting areas.

separation between header and data. These equivocal concepts mean that it would be impossible to correctly detect all areas. That said, we can use a heuristic to infer these areas.

The heuristic we use in EXCELERATE is adjacency, where we check for continuous regions. A cell is adjacent to another cell if the edges touch each other. This means that cells that 'touch' each other on the diagonal are not adjacent. Empty cells cannot be adjacent to any other cell. Using this adjacency heuristic, we are essentially dividing the *Spreadsheet* into groups of connected cells.

We construct an adjacency graph and find areas using the Hopcroft-Tarjan DFS algorithm [49]. More precisely, we use the following algorithm:

1. Convert the spreadsheet to a graph. Convert every cell in a *Spreadsheet* to a node in the graph, then for every node find the neighbours in the list by comparing their location.
2. Find the connected components using the Hopcroft-Tarjan DFS algorithm [49]. It outputs a set of connected nodes.
3. Create the areas. For every connected component, we construct a bounding box of the nodes in the connected component, which represents the Area.

When the areas are found, they are passed along to the next step of structure detection: the actual classification of the areas.

## Limitations of the heuristic

Spreadsheet 3.6 illustrates the power and limitations of the adjacency heuristic. The adjacency heuristic detects most structures that have empty rows, like the area `A3:E8` . This is due to computed columns that require formulas in all cells of the column, also when the other columns are empty. As such, the two regions `A3:E4` and `A6:E6` are connected using this formula in `E5` —essentially acting as a bridge.

However, when this is not present, such as the missing formula in `E9` , the two regions remain split. Furthermore, another failure happens when two tables are adjacent, such as the two tables in `G1:H9` .

To address these issues, other area-detection algorithms can be applied, or the adjacency graph can be adapted to include decorative formatting heuristics. For instance, if the background is the same color,

it can also be regarded as the same area. However, this would increase the false positives when the whole spreadsheet is a single colour.

### 3.3.3. Classifying Structures

Having identified the areas in the *Spreadsheet*, we can start classifying based on the content in the structures. Classification is a two-step process. First we determine if the structure meets the requirements based on several heuristics. If it complies, the literal structure is extracted to a *Structure* node in the *Compute Model*. In this section, we discuss the heuristics required for classification of the structures, also covering their limitations.

#### Heuristics

Before we discuss what the compiler sees as a *Table* or *Chain*, we first stress the fact that we work with heuristics. Just like with detecting areas, correctly identifying structures is hard. Detecting all edge cases is hard or even impossible to do. For instance, when a table only has strings in the first row, there is an ambiguity between the row being a header, or just a row with string data.

The heuristics we present here are not perfect and have their limitations. Furthermore, we require the structures to be bigger than a few rows to reduce the possibilities of false classifications. For *Tables* and *Rows*, this means that the actual data in the table (the rows between header and footer) has to span more than one row.

#### Tables

The characteristics of the table defined in Section 3.1.1 can be used to detect the structure. To reiterate, a table is a rectangular region with optional title, headers and footer. The rows of the table do not depend on each other.

We use the following conditions to identify *Tables* in a *Spreadsheet*. This rule-set is used as a heuristic instead of a formal specification, resulting in occasional error in classification. The key words “MUST”, “SHOULD”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119 [50]. The tables are detected according to the following rule-set:

1. A table MAY have a title. If the table has a title, it MUST be in the left-most cell of the first row and the rest of the first row MUST be empty.
2. A table MAY have a header. If the table has a title, the header MUST be in the second row. If the table does not have a title, the header MUST be in the first row. The header row MUST only contain string value cells.
3. A table MUST have more than one data row.
4. A data table column MUST have cells of the same type excluding empty cells.
5. A computed table column MUST only use references from the same row. All cells in the computed column MUST be the same formula, excluding the differences for row references. It should have the same ‘shape’.
6. A table MAY have a footer. If the table has a footer, it MUST be the last row of the table. The operations in the cells MUST be aggregation operations. If the first column is a data column, the first cell of the footer MAY be a string value cell (for the “TOTAL” text or similar).

If the area is determined a *Table* according to the rule-set, it is converted to an *Table* node in the *Structure Model*.

	A	B	C	D	E
1					
2	<b>Montly Expenses</b>				
3					
4	<b>Description</b>	<b>Category</b>	<b>Projected</b>	<b>Actual</b>	<b>Difference</b>
5	Extra... activities	Children	40.00	40.00	=D2-C2
6	Medical	Children			=D3-C3
7	School supplies	Children			=D4-C4
8	School tuition	Children	100.00	100.00	=D5-C5
9	Concerts	Entertainment	50.00	40.00	=D6-C6
10	Live theater	Entertainment	200.00	150.00	=D7-C7
11	Movies	Entertainment	50.00	28.00	=D8-C8
12	Music	Entertainment	50.00	30.00	=D9-C9
13					

Spreadsheet 3.7: An excerpt of the Montly expenses spreadsheet of the family budget workbook.

The area in Spreadsheet 3.7 is a table. The table has a header, which is in the first row (since the table does not have a title) and all of them are strings. When we have detected a title, headers and footers, it checks the columns of the rows in-between the header and footer to determine the type of the columns. In the case of Spreadsheet 3.7, there are four data columns and one computed-column.

We need to compare formulas to determine if a column is a computed column. To do this, we introduce the concept of the *shape* of a formula. The *shape* describes the relative positions of the dependencies of the cell. With the shape of a cell, we can check if a cell has the same relative dependencies as other cells in the same column. For instance, the computed column in Spreadsheet 3.8 has two cells with different shapes, since the relative dependencies are different: The C4 cell misses the extra + A3 at the end.

**Limitations.** One of the limitations of these rules is already discussed above: when the header row contains strings, it is unclear if this is data or the names of the headers. In this case, the compiler will interpret them as headers.

Another big limitation is the use of ‘external’ references: references outside the table. When this reference is constant, the shape of the formula between rows does not change, and we get the desired result. However, when a calculation references the rows in another table such as in Spreadsheet 3.9, the shape is different for every cell, and as such, the row is not recognised as a formula row.

## Chain

The chain as described in Section 3.1.1 can also be detected according to its characteristics. The chain table looks like a normal table but has the changed restriction that it allows for recursive columns, where a cell may depend on a cell in another row in the chain.

	A	B	C
1	Data Column 1	Data Column 2	‘Computed’ Column
2	<i>a</i>	<i>x</i>	= (4 * B2) + A2
3	<i>b</i>	<i>y</i>	= (4 * B3)

Spreadsheet 3.8: An example of a computed column with mismatching shapes.

	A	B	C	D	E	F	G
1							
2	<b>Table 1</b>						<b>Table 2</b>
3							
4	<b>Projected</b>	<b>Actual</b>	<b>Difference</b>				<b>Potential</b>
5	40.00	40.00	=B5-A5				=C5*1.005
6			=B6-A6				=C6*1.005
7			=B7-A7				=C7*1.005
8	100.00	100.00	=B8-A8				=C8*1.005
9	50.00	40.00	=B9-A9				=C9*1.005
10							

Spreadsheet 3.9: An excerpt of the Montly expenses spreadsheet of the family budget workbook.

We use the following conditions to identify *Chains* in a *Spreadsheet*. This rule-set is used as a heuristic instead of a formal specification, resulting in occasional error in classification. The key words “MUST”, “SHOULD”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119 [50]. The tables are detected according to the following rule-set:

1. A chain MAY have a title. If the chain has a title, it MUST be in a single cell in the first row and the rest of the row MUST be empty.
2. A chain MAY have a header. If the chain has a title, the header MUST be in the second row. If the chain does not have a title, the header MUST be in the first row. The header row MUST only contain string value cells.
3. A chain MUST have more than one data row.
4. A chain MUST have one or more initialisation rows.
5. A data chain column MUST have cells of the same type excluding empty cells.
6. A computed chain column MUST only use references from the same row. All cells in the computed column MUST be the same formula, excluding the differences for row references. It should have the same ‘shape’. All cells in the computed column MUST not reference a cell from a calculated chain column.
7. A recursive chain column MUST have all cells use references from the same row or the rows above it except for the base cases. The base cases MUST be in a higher row than recursive formula cells.. All recursive formula cells in the chain column must have the same shape.
8. A chain MUST have at least one recursive chain column.

Just like with a *Table*, we first check for titles, headers and footers, since they affect the range of the data. Then, we determine the type of columns of the *Chain* based on the contents of the column. Again, we use the *shape* of a cell to check if it is *Computed* or *Recursive*.

When a computed chain references a recursive chain column, it will automatically become an recursive chain column since we need the value of that cell, which can only be calculated recursively. Furthermore, a big difference between the table and the chain is the initialisation. We detect the initialisation by looking at the cell shape. When a column needs an initialisation, it will need a different shape for the base case and formula cells to calculate the rest.

A great example of the conversion of a chain is the savings spreadsheet, for which an excerpt can be found in Spreadsheet 3.10. Note that we find two areas: A1:D8 and F1:F2. Only A1:D8 is detected as a chain. We can see two recursive columns: *Interest* and *Total* which both reference the previous columns.

	A	B	C	D	E	F
1	<b>Date</b>	<b>Interest</b>	<b>Deposit</b>	<b>Total</b>		<b>Interest Rate</b>
2	01/03/2021			10000		0.015
3	01/04/2021	= D2 * F2	500	= D2 + B3 + C3		
4	01/05/2021	= D3 * F2	500	= D3 + B4 + C4		
5	01/06/2021	= D4 * F2	500	= D4 + B5 + C5		
6	01/07/2021	= D5 * F2	500	= D5 + B6 + C6		
7	01/08/2021	= D6 * F2	500	= D6 + B7 + C7		
8	01/09/2021	= D7 * F2	500	= D7 + B8 + C8		

Spreadsheet 3.10: An example of a chain that is correctly detected. The chain contains an initialisation row in row 2.

## 3.4. Embedding the Structures

Having extracted the structures out of the *Structural Model*, we need to incorporate them into the *Compute Model*. As we discussed in Section 3.2.3 the *Structural Model* does not change the formulae to account for the newly found structures. It is up to the *Compute Model* to update the *Compute Units*.

All references that reference the structures need to be updated to refer to the new structures. For instance, a range may be referring to a column of a table. Instead of a range, we would like to refer to this as a *Column Reference* that also references the table.

Doing this replacement in the graph can be computationally expensive. We would have to traverse the graph for every structure to make sure the dependencies are correctly linked to the structures. However, due to the grid-like nature of the structures, it is easier insert them into the *Compute Model* before we turn it into a graph. Hence, in this section, we introduce a new form of the *Compute Model* called the *Compute Grid*.

To accommodate for the structures, the *Compute Model* had to change slightly. In this section, we first introduce the *Compute Grid* and the required additions to the *Compute Model* in the form of new *Compute Units*. Then, we discuss how to populate the new compute grid and other changes made to the compilation pipeline.

### 3.4.1. Compute Grid

We introduce the *Compute Grid*: A sparse two-dimensional grid-like data structure that models the computation done in every individual cell. Essentially, the Compute Grid is the stepping stone between the structure model and the compute graph: it retains the grid-like structure of the structure model, while modelling the calculations according to the compute model. It allows for more granular compilation to the compute model, since we can first convert the cells to the compute model, then embed the structures into the *Compute Model*, and link the *Compute Units* afterwards.

Spreadsheet 3.11 shows an example conversion from the *Structural Model* to the *Compute Grid*. E9 is taken as output. During the conversion to the *Compute Grid*, we use the input to the compiler to adjust which cells are converted to the *Compute Grid*. As a result, the *Compute Grid* removes all cells that are not used by the calculations.

In order to support the *Compute Grid*, we need a way to express references to cells, ranges and structures in the *Compute Model*. As such, we need to update the model to allow for these nodes.

	A	B	C	D	E
1					
2	<b>Expenses</b>				
3					
4	<b>Description</b>	<b>Category</b>	<b>Projected</b>	<b>Actual</b>	<b>Difference</b>
5	Activities	Children	40.00	40.00	=D2-C2
6	Medical	Children			=D3-C3
7	School supplies	Children			=D4-C4
8	School tuition	Children	100.00	100.00	=D5-C5
9	<i>TOTAL</i>				=SUM(E5:E8)

	A	B	C	D	E
1					
2					
3					
4					
5			40.00	40.00	F('-', [D5, C5])
6			0	0	F('-', [D6, C6])
7			0	0	F('-', [D7, C7])
8			100.00	100.00	F('-', [D8, C8])
9					F('SUM', [E5,E6,E7,E8])

Spreadsheet 3.11: An example of the conversion of the *Structural Model* to the *Compute Grid* with E9 taken as output. Many redundant cells are removed. The F is a shorthand for a *Function Compute Unit*.

### 3.4.2. Compute Model

```

record ComputeUnit(Type type, Location Location, ComputeUnit[] Dependencies);
→ record Nil();
→ record ConstantValue(object Value);
→ record Input(string Name);
→ record Function(string Name);
++ → record GridReference()
++ → record CellReference(Location location)
++ → record RangeReference(Range range)
++ → record StructureColumnReference(string structureId, string columnName)
++ → record StructureCellReference(string structureId, string columnName, int index)
++ → record StructureFooterReference(string structureId, string columnName)
++
++ record Structure(string Id, Range Location)
++ → record Table(Column[] columns, TableData data)
++ → record Chain(Column[] columns, ChainData data)
++
++ record Column(string Name, ComputeUnit? Footer)
++ → record DataColumn()
++ → record ComputedColumn(ComputeUnit computation)
++ → record RecursiveColumn(ComputeUnit computation)
++
++ record StructureData(string structureId)
++ → record TableData(Dict<string, ComputeUnit[]> columns)
++ → record ChainData(Dict<string, ComputeUnit[]> dataColumns, Dict<string, ComputeUnit[]> baseCases)

```

Listing 3.2: Additions to the formal *Compute Model*.

The *Compute Model* is unable to express explicit references to other cells that we need in the *Compute Grid*. As such, we need to add nodes to the IR that support them. These nodes resemble the *References* in the *Formula Language*. We also add the structures and columns with references that directly point to structures.

## References

Within the *Compute Model*, references are used to denote a link to a *Compute Unit* in another cell of the *Compute Grid*. We distinguish between *Simple* and *Dynamic* references. *Simple* references are the same as the references in the *Formula Model*, they reference a static region on the grid, like *Cell References* and *Range References*. These nodes will be removed once we convert the *Compute Grid* to the *Compute Model*, since they can be replaced by the actual node(s). For instance, a *Cell Reference* can be replaced by the *Compute Unit* in the cell it references.

For structures, we need references that reference a structure instead of a location on the grid. We call them *Dynamic References*. These references can be to a column, a footer or an individual cell in the structure. They always contain the name of the structure and the column they refer to.

## Structures

The structures in the *Structure Model* need to be transformed to the *Compute Model*. Besides copying the name and range, this transformation primarily extracts the computations from the structures. The structures both have *Column* nodes that describe the type of column it is. Like the *Structure Model*, we distinguish between three column types: *Data*, *Computed*, and *Recursive*. The latter two use a *Compute Unit* to describe the repeated calculation.

**Structure Data.** The data of the columns is stored separately in a *Structure Data* node to maintain compute and data separation. When a structure is marked as input, this data does not have to be extracted. For non-input structures, only the data of the *Data Column* and the base cases of the *Recursive Columns* is extracted. The data is stored as a list of *Compute Units*, since the data does not have to be constant and may contain references to other *Compute Units*.

### 3.4.3. Changes in the Compiler Pipeline

The changes in the *Compute Model* allow for structures to be used. However, the current compiler pipeline does not utilise the structures from the *Structural Model*. As such, the steps in the *Compute Phase* has to be altered.

Currently, we employ type inference and input substitution. To adjust for the new structures, we remove the transformation step between the *Structural Model* and the *Compute Graph*. Instead, we start with a similar transformation from the *Structural Model* to the *Compute Grid*. Then, we insert the input as described in Section 2.3 and extract the *Structure Data*. Afterwards, we insert the *Structures* into the *Compute Grid*.

Then, having a fully constructed *Compute Grid*, we convert it to the *Compute Graph* where we infer types and do one last compiler pass to ensure all references correctly reference the structures.

In the next sections, we discuss the additions and changes made to the compiler.

#### Structure Model to Compute Grid

For the conversion of the *Structure Model* to the *Compute Model* we look at the outputs the user has specified and convert that cell from the *Structure Model* to a tree of *Compute Units*. For every *Value Cell*, we create a *Constant Value*. For every *Formula Cell*, the formula is transformed from the *Formula Model* to the *Compute Model*.

This tree of *Compute Units* is placed inside the grid. Then, we look at the referred location (in the *Spreadsheet*) of *Reference* nodes inside the *Compute Units* and convert those cells as well. This has the nice side-effect that cells that are not used in the final computation are not present in the *Compute Grid*.

The conversion of the *Structural Model* to the *Compute Grid* is a simplified version of the algorithm described in Section 2.3.2 that converts the *Structural Model* to the *Compute Graph*. The only difference is that we skip the last step: we do not link the converted *Compute Unit* to their parent.

**Structures.** The *Structures* in the *Structure Model* also get converted to the *Compute Grid*. This is done *after* the initial grid is constructed since we need it in constructing the *Structures*. For every structure in the *Structural Model*, all cells in a column are checked if they have any dependents (i.e. it is present in the *Compute Grid*). If the whole column is not present in the grid, it is pruned. The column type is always preserved.

#### Extracting Data

In the case that the user did not mark a structure as input, we need to capture the contents. This data is captured in a separate *Structure Data* object, separate from the compute grid and the structure. We

	A	B	C	D	E	F
1						
2				SCR(id, "Total", 0)		Constant(0.015)
3		SCR(id, "Interest", 1)	SCR(id, "Deposit", 1)	SCR(id, "Total", 1)		
4		SCR(id, "Interest", 2)	SCR(id, "Deposit", 2)	SCR(id, "Total", 2)		
5		SCR(id, "Interest", 3)	SCR(id, "Deposit", 3)	SCR(id, "Total", 3)		
6		SCR(id, "Interest", 4)	SCR(id, "Deposit", 4)	SCR(id, "Total", 4)		
7		SCR(id, "Interest", 5)	SCR(id, "Deposit", 5)	SCR(id, "Total", 5)		
8		SCR(id, "Interest", 6)	SCR(id, "Deposit", 6)	SCR(id, "Total", 6)		

Spreadsheet 3.12: An example of the from Spreadsheet 3.10 converted to the Compute Grid

store a reference to this object in the structure. As described before, the data is only extracted from non-computed and non-recursive columns.

The data is copied from the *Compute Grid* and stored as *Compute Units*. Since it is possible to reference data outside the structure through compute unit references. What data is stored depends on the structure. The *Table* only stores the data for the *Data Columns*. The *Chain* stores data for the *Data Columns* and the base cases for the *Recursive Columns*.

It is important that this compilation step is performed *before* we embed the structures since that step replaces the contents in the cell with references to the structures.

## Embedding Structures

When the data is successfully extracted and saved in the structure, we can replace the cells of the structure in the *Compute Grid* with references to the structure. Every cell in *Compute Grid* in the region of the *Structure* will be replaced with a reference. The cells in the columns are replaced by a *Structure Cell Reference*, referencing the structure and the column of the cell. The footers are replaced with *Structure Footer Reference*, referencing the footer operation in a column.

## Grid to Graph

In order to go from the *Compute Grid* to the *Compute Graph*, we use an algorithm not unlike the conversion of the *Structure Model* to the *Compute Grid*. We begin with the outputs and look at the compute unit tree in those cells. Then, for every reference we encounter in that tree, we link the *Compute Unit* in the referenced cell to the tree. As such, this is done in a recursive manner. In case of the range reference, we separately link all the roots of the cells in the range.

## Replacing References

While we already cover a lot with the inserting of the references in the compute grid, one reference remains broken: the range reference. When the range reference is compiled to the compute graph, we store all the compute units of the cells in the range as dependencies. When we compile this to code, we get a list of all the dependencies. If the *Range* references a random range in the graph, that is exactly what we want. However, when the *Range* references the column of a structure, we actually want it to be a special reference.

As such, the *Replace References* compiler step converts range references that reference a column or other reference of a structure to a *Structure Column Reference*. We traverse the graph and for every node, we look at its dependencies. If the dependencies are multiple *Structure Cell References*, we check if these cell references cover the whole column. If they do, we convert these *Structure Cell References* to a single *Structure Column Reference*.

More importantly, this pass also introduces the *creation* of the structures. Until now, we have always assumed that structures always existed. However, when compiling this to code, we need to instantiate this structure. As such, we create a *Structure Creation* Compute Unit. Every reference to a structure will have a dependency on the creation of that structure.

In this pass, we also introduce the dependencies of the *Structure Creation* itself. The dependencies are the data of the structure, stored in the *Structure Data* object. The calculations for this data is also inserted into the Compute Graph, taking existing nodes into account.

## 3.5. Coding the Structures

Once the *Compute Model* has been enriched with the structures and the *Compute Graph* has been optimised to support the structures, we can output the structures and make the code more readable. The *Code Model* is already expressive enough to support this new addition to the *Code Phase*, so we only need some changes to the compilation flow.

First, we need to create the actual types and constructors, given the *Structures* and *Structure Data* respectively. Then, the newly created *Classes* need an optimisation to handle mutual recursion that otherwise significantly slows the compiled code. In this section, we will discuss these changes.

### 3.5.1. Creating types from structures

In the *Compute Model*, structures were separately stored. In the *Code Model*, they are still stored separately, but will be compiled to a *Class*. This *Class* will be instantiated in the main code, and operations will be called to calculate values in the *Structure*. In this subsection, we quickly discuss what types are created for the *Table* and *Chain* types.

	A	B	C	D	E
1					
2	<b>Montly Expenses</b>				
3					
4	<b>Description</b>	<b>Category</b>	<b>Projected</b>	<b>Actual</b>	<b>Difference</b>
5	Extra... activities	Children	40.00	40.00	=D2-C2
6	Medical	Children			=D3-C3
7	School supplies	Children			=D4-C4
8	School tuition	Children	100.00	100.00	=D5-C5
9	<i>TOTAL</i>				=SUM(E5:E8)

Spreadsheet 3.13: The *Structure Model* of a spreadsheet containing a *Table* describing the monthly expenses. Taking E9 as output, the Description and Category columns will be pruned.

### Tables

Since a table is defined as independent rows, the *Code Model* sees it as a list of row data. This row data is transformed to a 'table item' class. All columns are present as properties of the class. The data columns are normal properties, and the computed columns will become computed properties. The *Table* itself is represented as a list of that 'table item'.

Take Spreadsheet 3.13 for instance. This *Table* will contain three columns when converted to the *Compute Model*: the two data columns `Projected` and `Actual` and the computed column `Difference`. Compiling this *Table* to the *Code Model* will result in a new type or class called `MonthlyExpensesItem`.

containing the properties `ActualCost`, `ProjectedCost`, and the computed property `Difference` which just calculates the difference depending on the other properties.

```
class MonthlyExpensesItem
{
    public double ProjectedCost { get; set; }
    public double ActualCost { get; set; }
    public double Difference => ProjectedCost - ActualCost;

    public MonthlyExpensesItem(double projectedCost, double actualCost)
    {
        ProjectedCost = projectedCost;
        ActualCost = actualCost;
    }
}
```

Listing 3.3: The compiled code of the *Table* in Spreadsheet 3.13. The `Difference` computed column is created as a computed property.

The actual table is represented by the list of all items: `List<MonthlyExpensesItem>`. Operations on the columns of the table will be done through mapping operations, employing LINQ in C#. For instance, to get the sum of all the differences in the `monthlyExpenses` table, we say `monthlyExpenses.Sum(m => m.Difference)`. In other languages, this would have nearly the same syntax, such as the map operation in Kotlin: `monthlyExpenses.sumOf { it.Difference }`.

The construction of the *Table* is done through the creation of a list, with creation of the individual items representing the rows. For the example above, the constructor would look like this:

```
List<MonthlyExpensesItem> = [
    new MonthlyExpensesItem(100, 105),
    new MonthlyExpensesItem(90, 50),
    ...
];
```

When a structure is not marked as input, this constructor is called in the main function. The data is sourced from the *Table Data*.

## Chains

Chains are a bit more complicated than Tables. Because the rows are depending on each other, we cannot exercise the same strategy as with the tables. As such, we represent the chain as its own class. Data columns are stored column-oriented instead of row-oriented. Computed and Recursive columns are calculated on an individual basis: In order to calculate a value in a cell in a computed or recursive column, we call the function `{ColumnName}At(x)`.

For instance, the savings chain in the budget example is compiled to the following code:

```

public class Savings
{
    public List<double> Deposit { get; set; }

    public double TotalBaseCase {get; set;}

    public double InterestAt(int counter) => 0.015 / 12 * TotalAt(counter - 1);

    public double TotalAt(int counter)
    {
        if (counter == 0) return TotalBaseCase;
        return TotalAt(counter - 1) + InterestAt(counter) + Deposit[counter - 1];
    }

    public Savings(List<double> deposit, double totalBaseCase)
    {
        Deposit = deposit;
        TotalBaseCase = totalBaseCase;
    }
}

```

Listing 3.4: A code snippet of the Savings chain compiled to C# code.

The *Deposit* column is compiled to a list of doubles. Furthermore, the computed and recursive columns are compiled to the `InterestAt` and `TotalAt` properties. Notice that the `TotalAt` property contains a recursive call to itself, and thus requires a base case. The computed column *Interest* does not require a base case since it does not have a reference to itself.

The construction of the *Chain* is done differently than the table. Instead of creating individual items, we pass each column along as input:

```
Savings savings = new Savings([500, 550, 450], 10_000)
```

When a structure is not marked as input, their constructor is called in the main function. The data is sourced from the *Structure Data*.

### 3.5.2. Optimisations for Mutual Recursion

Running the emitted code that includes the chain Listing 3.4 without optimisations leads to a very long runtime. It appears that the conversion to imperative code introduced an unwanted performance hit. Upon closer inspection, this is due to the recursive nature of the chain.

The two recursive definitions call each other and as such, the functions have *Mutual Recursion*. In the example, we see that the `TotalAt` calls itself *and* the `InterestAt` function. However, the `InterestAt` function also calls the `TotalAt` function. This means that one call to the `TotalAt` call creates two extra calls to `TotalAt`. Hence, the amount of times `TotalAt` is called equals  $2^x$  where  $x$  is the input to the function.

There are some ways to fix mutual recursion, one of which is to transform mutual recursion to single recursion [51]. This is done through inlining of one of the methods. However, this is not really desirable, since it will lead to duplicate code, and that is exactly what we are trying to avoid.

Instead, we use memoization. Memoization is the practice of storing the input and output of a pure function as a pair [52]. When a function is executed, the input is checked against that pair, and if there is a matching pair that output is returned, saving on the computation costs and avoiding the spawning of further recursive calls [52]. On the first call with a new input, the output is calculated and added to the list of pairs.

We do this with the mutual recursion in Listing 3.4 as well. Listing 3.5 shows the new mutual recursion solution. Every function that spawns two or more function calls to itself (direct or indirect) will be memoized. As such, `InterestAt` will not use memoization, but `TotalAt` will. This speeds up the execution by orders of magnitude.

```

public class Savings
{
    private readonly Dictionary<int,double> _totalAtMemoization = new();

    public List<double> Deposit { get; set; }

    public double InterestAt(int counter) => 0.015 / 12 * TotalAt(counter - 1);

    public double TotalAt(int counter)
    {
        if (_totalAtMemoization.ContainsKey(counter))
            return _totalAtMemoization[key];

        if (counter == 0) return 10000;

        double result = TotalAt(counter - 1) + InterestAt(counter - 0) + Deposit[counter - 1];

        _totalAtMemoization.Add(key, result);
        return result;
    }

    public Savings(List<double> deposit)
    {
        Deposit = deposit;
    }
}

```

Listing 3.5: A simplified code snippet of the Savings chain compiled to C# code.

## 3.6. Discussion of the Compiler Design

During development of the compiler, several design decisions shaped the pipeline that is presented in the previous and this chapter. While we have already deliberated the design choices on the existing architecture, some choices that were made were hidden or did not fit the narrative of the sections. In this section, we discuss these decisions.

### 3.6.1. Data Model

The original architecture contained a dedicated *Data Model* and *Data Phase* that would run parallel to the *Compute Phase*. The *Data Model* would describe the concrete values that resided in the structures and was based on the data layout dialect in MLIR [53]. The original idea was that this would cause complete separation between data and compute. However, this layer proved counter-productive for several reasons.

First, the *Data Model* prevents values from being computed. Because of the complete separation, values cannot be represented by a *Compute Unit*. As such, it was impossible to refer to another cell from within a data column, since the *Data Model* could not understand this. This was especially problematic in chains, since they often contain references in their base cases.

Furthermore, the new *Structure Data* concept provides the same descriptive power, including computation, while still separating the data from the definition of the structure.

Finally, the removal of the *Data Model* reduced the compilation flow from four phases—with two running in parallel—to a much simpler three-phase sequence. It also means that there are no more mutual dependencies between two compiler phases. This considerably lowers the cognitive load and improves maintainability.

### 3.6.2. Computed Properties

The current implementation of structures and their computed properties was implemented in a simple way. This is effective for most use-cases, but introduces some limitations on the compiler: A computed property cannot have an external input as reference. A computed property is currently evaluated in

isolation of the global scope. This means that the input is not available and as such, the emitted code will not compile. This represents a potential threat to validity that could be addressed in a future redesign of structures.

A possible redesign would be to rethink the structure design. Each externally referenced cell would be collected and stored separately. When emitting the code, the classes representing the structure would include these external references as properties of the class and they would get their value from constructor parameters.

This poses a limitation on the list, since it is not a class and cannot add. Instead of a list with items, the table would be its own type, just like the chain. Once we have parity, we can introduce these new properties. Due to limited time, this more robust system was not implemented.

***Merging structures.*** Due to this limitation in the current compiler, it is not possible to reference data in another structure. This enforces isolation between structures. As a result, when a table 'shares' a column through formula dependencies with another column, the compiler will not compile the spreadsheet correctly.

*Chapter 4*  
**Evaluation**

**4.1. Methods ..... 55**  
**4.2. Results ..... 59**  
**4.3. Discussion ..... 61**



A compiler cannot be fully assessed on its design considerations alone. Its effectiveness must also be demonstrated through systematic evaluation. For EXCELERATE, this involves the verification of the translation from spreadsheet formulas to C#. The spreadsheet should produce the same values as its compiled C# counterpart.

The proposed evaluation in this chapter serves two purposes: to test the correctness of the compiler, and to assess the qualitative aspects of the generated code including performance, readability and adherence to idiomatic C# practices.

As such, this chapter reports on the evaluation along multiple factors. We first introduce the methods and metrics in Section 4.1. Then, we show the results of the experiments in semantic equality and performance in Section 4.2. Furthermore, we discuss the readability of the code through several examples. Finally, we consolidate the findings in a discussion in Section 4.3, also denoting the threats to validity of the research and experiments.

## 4.1. Methods

In this subsection, we propose several techniques to evaluate our Excel compiler. First, we discuss a way to analyse the semantics and performance of the compiled spreadsheet. Then, we discuss our method of assessing the readability of the generated code.

### 4.1.1. Semantics and Performance

In our first experiment, we test the semantics and performance of EXCELERATE through a simple differential test [54]. Essentially, we generate random input for all eligible cells and compute the value from those inputs using both the Excel calculation engine, and our compiled spreadsheet code. We compare the inputs for equality and compare the performance of both methods. In the next subsections, we will discuss this more thoroughly.

#### Spreadsheet Selection

We select the spreadsheets from the Microsoft Create Excel Templates repository. This repository contains spreadsheets that are well-formatted and widely used. We select spreadsheets based on the compatibility with the compiler. Most spreadsheets in the repository contain formulas unsupported by the compiler. For instance, spreadsheets may contain error handling logic for invalid inputs, which are functions the compiler does not support, such as `IsError()`. The selected spreadsheets do not contain these functions, or we remove them by adapting the spreadsheet to the compiler.

Furthermore, we fix some dynamic aspects of a spreadsheet by removing the `Index` and similar functions. Functions like the `Index` function return a range and are mostly used for creating dynamic ranges that respond to user input. The compiler does not support this, and thus we replace this dynamic aspect by replacing these functions with a static range.

Ultimately, we selected and adapted 5 spreadsheets that represent different use-cases and sizes. Four spreadsheets are found in the Microsoft Create Excel Templates repository, and one spreadsheet is taken directly from an actuarial context. Table 4.1 provides an overview of the spreadsheets.

Name	Cells / CU	Structures	Inputs	Description
Monthly Budget	320 / 701	2 Tables, 1 Chain	3 (3 ranges)	Computationally the most diverse workbook, containing tables describing incomes and expenses and a way to calculate interest on a savings account.
Holiday Budget	83 / 85	6 Tables	6 (6 range)	Describes the budget of a holiday with six tables.
Service Invoice	30 / 53	1 Table	2 (1 range + 1 cell)	A very simple workbook that describes an invoice with one table containing services purchased.
Retirement Planner	1049 / 5733	1 Chain	3 (1 range + 2 cell)	A workbook describing the interest on a bank account with monthly withdrawals for pension planning.
Actuarial Example	705 / 3007	2 Tables	2 (2 ranges)	An example workbook containing actuarial calculations, provided by DNB (De Nederlandse Bank). This spreadsheet is more than 150Mb.

Table 4.1: An overview of the Spreadsheets used in the experiments. The table describes the name and gives a description of their contents. We also denote the amount of cells, compute units, structures and input to give a sense of the complexity of a spreadsheet.

## Trial Creation

Not every cell can be an input. Hence, we manually determine what cells are inputs. Based on the limitations discussed in the previous chapter, we exclude cells that would break the compilation. For instance, references made to computed cells from structures cannot be an input. Furthermore, only numerical values are considered to be eligible.

Every workbook gets assigned one output cell that ensures that nearly all parts of a workbook are covered by the experiment. In other words, the calculations that compute the output cell preferably reference the whole workbook.

For every workbook, we consider all eligible cells as input. For every eligible cell, we compute a random value that is used. We call the collection of the output and all these random input values for a spreadsheet a *Trial*. For instance, a *Trial* for the ‘Monthly budget report’ spreadsheet includes random input values for the income, the expenses and the deposits. For EXCELERATE compiled code, we use emitted structures if they are available.

The *Trial* is associated with a spreadsheet. This spreadsheet gets compiled with the eligible input cells and output as parameters. This creates a class library that is referenced and directly called by the evaluation script.

## Running the Trials

To compare the methods, we require test runners or *evaluators* for EXCELERATE and the spreadsheet in Excel. The EXCELERATE evaluator references the compiled class library and calls it directly with the generated input for any given *Trial*. The output is saved for future reference in a *Trial Result* object.

Running the trials with the Excel calculation engine is more complex since we need the exact Excel application, not an external library. Hence, we utilise a part of the Microsoft Office Suite called the *Interop*. This enables us to alter a spreadsheet using C# and evaluate formulas using the original calculation engine. Every spreadsheet is loaded using this method and the inputs of the *Trial* are copied into the spreadsheet. Using a C# method provided by the *Interop*, we calculate the value in the output cell and append the *Trial Result* object with this value.

Using the *Interop* comes at a cost since we need to transfer the data between the Excel application and the C# evaluation code. This is done through the .NET COM interface, which involves moving data between processes. This introduces a non-deterministic overhead that is required to run Excel from C#.

## Comparing the methods

We compare the methods along two metrics:

1. semantic equality through the calculated outputs and
2. performance on several components through timers in the evaluation script.

**Semantic Equality.** The semantic equality is validated by comparing the outputs of both evaluators. For every *Trial Result* object, the output of both evaluators should be the same, while accounting for floating point imprecisions. To test this, we use the following formula:

$$|x_{\text{Excelerate}} - x_{\text{Excel}}| \leq \varepsilon$$

where  $\varepsilon = 10^{-6}$  is a very small value that accounts for floating point imprecisions.

The semantic equality of EXCELERATE is measured as an *Equality Rate* (ER)

$$\text{ER} = \frac{c}{n}$$

which is expressed as a fraction of the number of correctly matching outputs  $c$  over the total number of comparisons  $n$ .

**Performance.** Furthermore, the performance is recorded on several components. Through our preliminary tests, we noticed that the inserting and extraction of data in Excel took longer than anticipated. Hence, we also recorded this component resulting in the following components being measured:

$$t_{\text{wall}} = t_i + t_c + t_e$$

where

- $t_{\text{wall}}$  is the actual time it takes to complete the experiment.
- $t_i$  is the *Injection Time*: The time it takes to inject the values into the Spreadsheet, measured by the time it takes to transform the values into the desired object and assigning them to the *Interop* Excel worksheet object.
- $t_c$  is the *Calculation Time*: The time it takes for EXCELERATE to run the emitted main method or Excel to finish calculating the cells in the whole spreadsheet.
- $t_e$  is the *Extraction Time*: The time it takes to extract the values out of the spreadsheet, measured by the time it takes to read the output cell property of the *Interop* Excel worksheet object.

When measuring performance, we exclude outliers. These outliers are often caused by demanding background processes and are not representative samples for the methods. A point is an outlier when the value lies more than  $1.5 \times \text{IQR}$  from the closest quarter  $Q_1$  or  $Q_3$ . This should decrease variability and produce samples that are representative of the methods. In order to assess variability, we ran repeated experiments. Every spreadsheet was evaluated 100 000 times with random input. Durations are measured using the .NET Diagnostics `Stopwatch` class, which uses a high-resolution performance counter to achieve sub-microsecond precision (100ns) [55]. However, due to the long nature of the experiments, we present the duration in milliseconds.

## Setup

All experiments have been done on a desktop computer with a 13th Gen Intel(R) Core(TM) i7-13700 and 32GB DDR5 RAM. The system was set to high-performance mode and a program was running to prevent the computer from going into sleep mode.

**Variability.** To capture the variability of the results, we report the mean and the standard error. For a series of samples:  $(x_1, x_2, \dots, x_n)$  we calculate the following:

$$\bar{X} = \frac{1}{n} \sum_{i=1}^n x_i$$
$$\sigma = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{X})^2}$$
$$SE = \frac{\sigma}{\sqrt{n}}$$

### 4.1.2. Readability

The second experiment assesses the readability of the code emitted by the Excel compiler. This is a subjective assessment, since judging the readability of code is hard to do objectively as we covered in Section 1.2.4. The main objective of this experiment is to showcase the improvements in readability between the ‘basic’ compiler and EXCELERATE. In the next sections, we briefly discuss how to obtain this code and what criteria we use for discussing the readability, based on Section 1.2.4.

### Preparation

Obtaining the code is done in two ways. We obtain the code for EXCELERATE by running the compiler for every spreadsheet, generating a project and storing the files. For the ‘basic’ compiler, we run the same compiler, but without the steps for structure detection as described in Section 3.2. Essentially, we force that no structure is found by clearing the array of structures in the *Structure Model* at the end of the *Structure Phase*. This forces EXCELERATE to fall back on the ‘basic’ compiler, compiling the code without structural guidance.

We use the same spreadsheets as in Table 4.1, as they represent a wide range of use-cases.

### Criteria

Before we present the results and discuss the readability, we first reflect on the criteria of ‘readable’ code. As we covered in Section 1.2.4, we consider this highly subjective metric to be good if the code seems to be written by an experienced developer in that particular language. This means that the code should be:

- *Expressive* and *Comprehensible*: It should be very simple to figure out what the program does. This can be communicated through the use of variable names or smart use of language features such as LINQ.
- *Extensible* and *Maintainable*: Since EXCELERATE is meant to be used as a class library, it should be easy to build programs on top of it. If a change needs to be made to the code, it needs to be easy to fix this. This also means that code should express a calculation only once and avoid code duplication.
- In accordance with the style guide: we use the Microsoft Style Guide for C# [39] for this. All C# code should adhere to this style guide. For instance, it states naming conventions for local variables, classes, properties, etc; which language constructs we should use and more.

Spreadsheet	Excel			EXCELERATE	Speedup
	Insertion	Calculation	Extraction	Calculation	
Monthly Budget	$9.07 \times 10^3 \pm 6.75 \times 10^0$	$4.26 \times 10^3 \pm 2.68 \times 10^0$	$3.18 \times 10^3 \pm 2.39 \times 10^0$	$5.12 \times 10^0 \pm 1.52 \times 10^{-2}$	$8.32 \times 10^2$
Holiday Budget	$2.07 \times 10^4 \pm 1.33 \times 10^1$	$2.32 \times 10^3 \pm 3.27 \times 10^0$	$3.70 \times 10^3 \pm 3.16 \times 10^0$	$2.65 \times 10^0 \pm 3.75 \times 10^{-3}$	$8.75 \times 10^2$
Service invoice	$1.07 \times 10^4 \pm 8.84 \times 10^0$	$2.57 \times 10^3 \pm 4.80 \times 10^0$	$4.18 \times 10^3 \pm 6.34 \times 10^0$	$4.07 \times 10^{-1} \pm 4.28 \times 10^{-2}$	$6.32 \times 10^3$
Retirement planner	$1.01 \times 10^4 \pm 8.36 \times 10^0$	$1.26 \times 10^4 \pm 4.43 \times 10^0$	$3.25 \times 10^3 \pm 1.92 \times 10^0$	$2.25 \times 10^1 \pm 3.73 \times 10^{-2}$	$5.60 \times 10^2$
Actuarial Example	$5.75 \times 10^3 \pm 4.71 \times 10^0$	$8.24 \times 10^3 \pm 6.26 \times 10^0$	$2.65 \times 10^3 \pm 2.17 \times 10^0$	$1.45 \times 10^0 \pm 2.00 \times 10^{-3}$	$5.68 \times 10^3$

Table 4.2: Overview of the experiment results that measure performance of EXCELERATE and Excel in microseconds ( $\mu s$ ). Results are the average of  $n = 100000$ . We also show the standard error.

## 4.2. Results

For the results, we briefly discuss the results for Semantic Equality. Then, we dive deeper into the results of the performance aspect of the experiments, showing the differences between the two programs.

### 4.2.1. Semantic Equality

All spreadsheets passed this experiments. For every spreadsheet, for all  $n = 100000$  experiments, every output comparison satisfied the defined equation with  $\varepsilon = 10^{-6}$ , resulting in an Equality Rate of 1.0 across all spreadsheets. A large fraction (>99%) of the results were bit-wise identical. The few differing results could still be accounted to floating point imprecision.

### 4.2.2. Performance

The Execution time was recorded for the compiled C# code and the Excel evaluation for  $n = 100000$  invocations per spreadsheet. Each component of the execution time was recorded separately for each invocation. The results can be seen in Table 4.2.

EXCELERATE manages to achieve microsecond performance on many spreadsheets, while Excel takes around 2 milliseconds for the calculations. The Excel calculations take considerably longer than

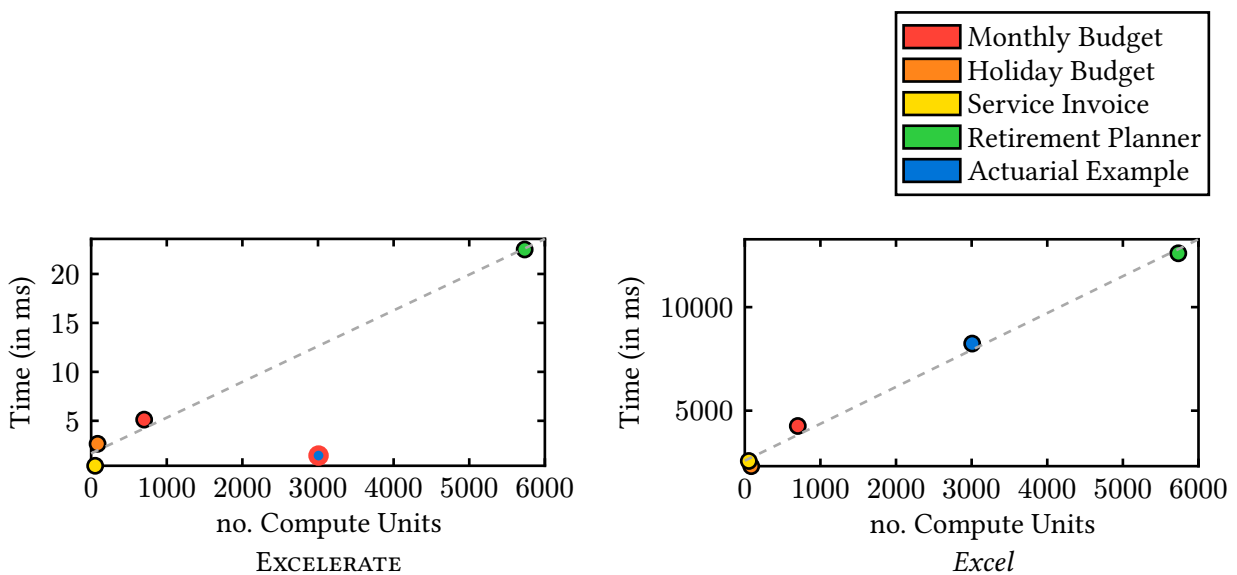


Figure 4.1: A scatter plot of the completion time versus the number of compute units for each spreadsheet in EXCELERATE emitted code and Excel. Trendlines have been drawn to show the relationship. The *Actuarial Example* is considered an outlier.

EXCELERATE. Table 4.2 shows the average speedup for every spreadsheet. The overhead of the COM interface is included in these figures. Taking the geometric mean, an average speedup of 1710x was observed. However, this includes the extreme speedups for both the simple *Service Invoice* and *Actuarial Example*. The *Service Invoice* is extremely simple, and investigating the *Actuarial Example*, we see that the chain that is present within the spreadsheet was not detected by EXCELERATE and as such, very verbose code was created.

Figure 4.1 shows the number of compute units against the time it takes to calculate for EXCELERATE and Excel. Apart from the *Actuarial Example* (blue) spreadsheet, the relationship between compute time and the complexity seems linear. The plots in Figure 4.1 are similar, indicating that both methods handle the complexity in the same way. Based on the speedup values in Table 4.2 and the linear trends in Figure 4.1, we infer a negative relationship between the speed-up and the complexity of the spreadsheet.

An interesting observation can be made for the *warmup* for the EXCELERATE compiler. The first run of the EXCELERATE compiler is often 250 times slower than the next run due to the overhead of the JIT compiler. We do not include this in the performance measures since it is only the first run and it significantly skews the variability.

Figure 4.2 shows the amount of time it takes to execute 100 000 invocations of the smallest spreadsheet: *Service invoice*. EXCELERATE is considerably faster than Excel, finishing compilation and calculation in the same time that calculation is completed. The compilation is about 85ms, accounting for 66.7% of the total wall time. Conversely, if we look at the composition of the performance of Excel, the insertion of data into the spreadsheet takes the longest, accounting for 61.3% of the total wall time. The actual calculation only takes 14.7% of the total wall time.

The insertion times for *Family Monthly Budget* and *Expense Tracker* are significantly higher than the calculation and extraction combined. Table 4.1 describes the amount of insertions needed. The amount of time spent on insertions linearly correlate with the cost of insertions. This cost of insertions is

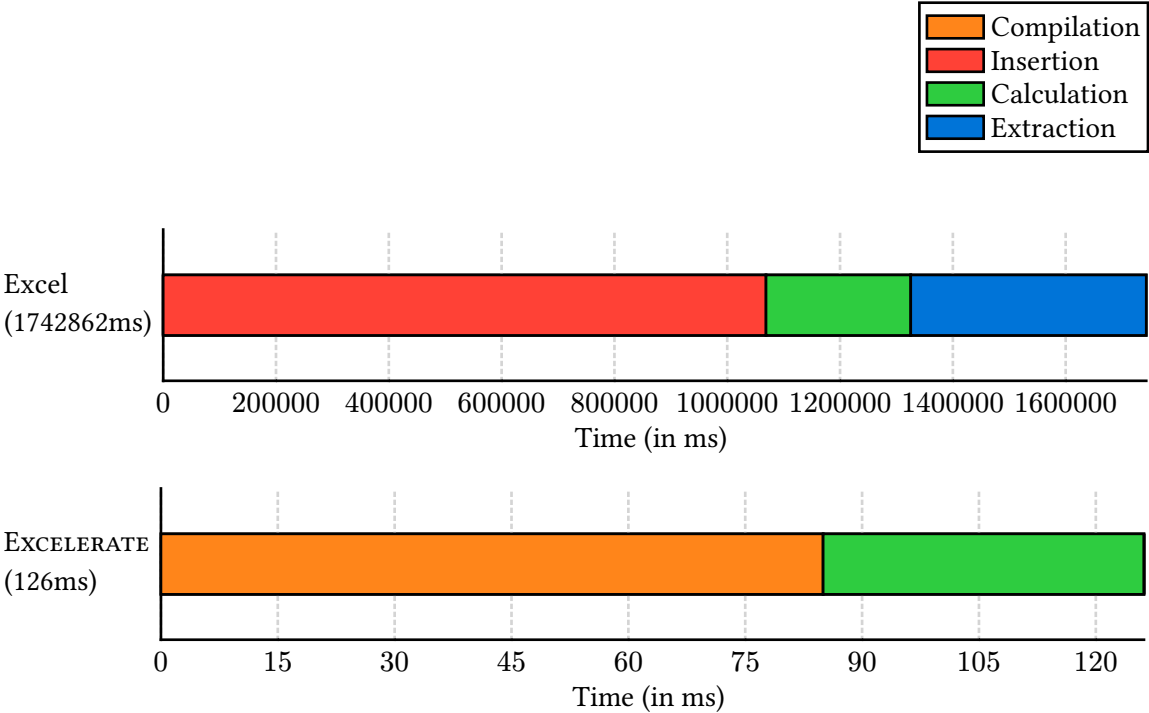


Figure 4.2: A visualisation of the total amount of time it takes to execute 100000 invocations on the *Service Invoice* spreadsheet. The *Compilation* step only has to be run once.

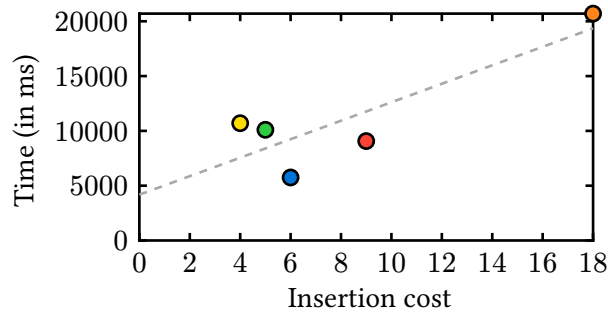


Figure 4.3: A scatter plot of the amount of time it takes to insert the input into the Excel spreadsheet for  $n = 100000$ . A linear regression line has been drawn. This indicates a linear relationship. The insertion cost is calculated as 3 units for ranges and 1 unit for cells.

chosen over simply taking the amount of insertions. Since ranges are more expensive to insert, as they need to transport more data, we count them as three cells.

### 4.3. Discussion

In this section, we discuss the results described in Section 4.2. We also discuss the readability of the compiled code based on compiled snippets.

The results show a reported total equality across all benchmark spreadsheets. This indicates that the compiled C# code reproduce the exact behaviour of Excel’s calculation engine. The findings directly answer RQ3, demonstrating that there is a way to verify semantic equivalence between Excel and generated code for the supported feature set. More precisely, the bit-wise equivalence found in many compared results indicates EXCELERATE often produces the same order of operations as Excel.

Our performance measurements indicate an average speedup of 1710x. The main contributor to this speedup is the removal of the COM interface: EXCELERATE can be directly called from C# code. However, this is not the only contribution. It can be seen that in larger spreadsheets such as *Family budget monthly*, the actual calculation time doubles in comparison with a smaller spreadsheet such as *Service Invoice*. The overhead of the COM interface should only interfere when data and commands are moving between program boundaries, not when the actual Excel sheet is calculating. This indicates that the Excel Calculation engine is also performing slower than EXCELERATES compiled code.

This can be explained by the optimised execution path. The Excel Calculation engine is an interpreter. Conversely, the code EXCELERATE emits is compiled ahead-of-time and further optimised by the .NET JIT compiler, significantly boosting performance [56]. Although the .NET compiler compiles and interpretes the code in IL, sections of the code are converted on-demand to machine code and directly run afterwards [56]. This gives a small overhead the first time we call a function for example. This is what we saw as the JIT warmup in Section 4.2.

The relationship between the complexity of the spreadsheet (measured in the number of Compute Units) appears to be linear for both Excel and EXCELERATE. Based on Figure 4.1, the outlier is the *Actuarial Example* spreadsheet. For EXCELERATE this speedup can be explained: since the chain in the spreadsheet was not correctly detected due to limitations discussed in Section 3.6, the code is very verbose. While this reduces readability, the compiled code can be much better optimised as the chain calculation is effectively completely inlined. Consequently, the code executes much faster.

The performance measurement revealed that instead of calculation, the insertion of data is the biggest slowing factor for Excel. Due to the large overhead of the COM interface, every time a outside program tries to insert a range of value, it gets this COM overhead, quickly accumulating when having multiple

input ranges. Instead, EXCELERATE produces a class library with little to no input and output extraction overhead.

The increases in performance and evidence for semantic equality indicate EXCELERATE as a possible substitution for the use of Excel in business critical applications. Applying Excel to the example in the introduction, the two weeks it took to calculate the pension prediction for all 200 000 customers, would take approximately 5 minutes with EXCELERATE. That said, the limited operator and function set supported by EXCELERATE is a critical limiting factor. However, EXCELERATE can be extended to support more operators, functions and paradigms.

Furthermore, having compiled code instead of interpreted code with Excel does not only have performance advantages. EXCELERATE itself is language agnostic and can compile to many languages if extended. This allows the spreadsheet to also be compiled to code that can be run on devices that do not have Excel installed. Besides, compiled code is also easier to audit and is more easily tested using automated tests. This is important for financial companies such as the pension funds.

### 4.3.1. Readability and Idiomaticity

In this section, we show the differences between the ‘basic’ compiler and EXCELERATE. Based on different spreadsheets, we compare the old and new code for improvements in readability and idiomaticity, based on the definition in Section 1.2.4 and Section 4.1.2. The full files can be found in the appendix.

#### Structural Abstraction

From a readability and comprehensibility standpoint, EXCELERATE code bundles common logic into classes that communicate their meaning directly. This removes the scattered variables that convey nothing about their semantics.

```
Exceleerate > Main()
1 List<TBL_MonthlyExpensesItem> tBL_MonthlyExpenses = [new(40, 40), new(0, 0), ..., new(0, 0), new(450, 450)];
2 double monthlyBudgetReportE8 = tBL_MonthlyExpenses.Select(t => t.ActualCost).Sum();
3 double monthlyBudgetReportD8 = tBL_MonthlyExpenses.Select(t => t.ProjecteCost).Sum();

Basic Compiler > Main()
1 List<double> monthlyBudgetReportE8List =
2 [
3     40,
4     0,
5     ...
6     0,
7     450
8 ];
9 double monthlyBudgetReportE8 = monthlyBudgetReportE8List.Sum();
10 List<double> monthlyBudgetReportD8List =
11 [
12     40,
13     0,
14     ...
15     0,
16     450
17 ];
18 double monthlyBudgetReportD8 = monthlyBudgetReportD8List.Sum();
```

Figure 4.4: A comparison listing, comparing the creation and calculation of the monthly expenses. EXCELERATE is the top snippet, and the Basic Compiler is the bottom snippet.

The use of LINQ extends this. For instance, calls that compute the sum of a column with `Sum` and `Select` clearly state what part of an entity is being processed, and how it is being done. In comparison with the basic compiler in Figure 4.4, where it is just a `Sum` on a variable with no real meaning, this increases the comprehensibility of the code.

```

Excelerate > Main()
1 public double Main(List<TBL_MonthlyExpensesItem> tBL_MonthlyExpenses)
2 {
3 // Used in the code

Basic Compiler > Main()
1 public double Main(double tBL_MonthlyExpensesE1, double tBL_MonthlyExpensesE2, double tBL_MonthlyExpensesE3, ...)
2 {
3 // Used in the code

```

Figure 4.5: A comparison between EXCELERATE and the Basic Compiler in terms of parameters. EXCELERATE bundles the parameters in a class, while the Basic Compiler needs to express all variables explicitly.

A side effect of this structural abstraction is significant less lines of codes and application of the DRY principle (see Section 1.2.4). In the EXCELERATE emitted code in Figure 4.4, the columns of the ‘Monthly Expenses’ now get merged into one construction instead of the two separate declarations in the old code. Furthermore, the main repetition that calculates the values in the interest spreadsheet is abstracted through a new class that recursively calculates. This improves maintainability and readability.

This new object-oriented design makes integration easier. EXCELERATE exposes groupings of variables by structure as input, allowing external code to easily populate these objects and call the generated code. This is an improvement over the old code, which only supported individual variables. This further improves readability as can be seen in Figure 4.5.

However, when the compiler does not recognise a single structure, the readability can decline fast. Figure 4.6 shows the code for emitted by EXCELERATE. While it makes use of the table structures found in another spreadsheet (as indicated by the different type names), it failed to recognise the chain structure. As such, even though it uses these new structures, its readability has not improved.

```

Excelerate > Actuarial Example
1 public double Main(List<TableRenteparameter_Psi_RA1C100Item> renteparameter_Psi_RA1C100,
2 List<TableRenteparameter_phi_R_NLA1D100Item> renteparameter_phi_R_NLA1D100)
3 {
4 double renteparameter_phi_R_NLB1 = renteparameter_phi_R_NLA1D100[0].Column1;
5 double renteparameter_Psi_RA1 = renteparameter_Psi_RA1C100[0].Column0;
6 double renteparameter_Psi_RB1 = renteparameter_Psi_RA1C100[0].Column1;
7 double renteparameter_Psi_RC1 = renteparameter_Psi_RA1C100[0].Column2;
8 double voorbeeldRTSE4 = Math.Exp(-1) / (1) * (renteparameter_phi_R_NLB1 + 0.03296211605999014 *
9 (renteparameter_Psi_RA1) + 0.014349731117662046 * (renteparameter_Psi_RB1) + 0.010609776508980583 *
10 (renteparameter_Psi_RC1)) - (1);
11 double voorbeeldRTSF4 = 100000 * (1 + voorbeeldRTSE4);
12 double renteparameter_phi_R_NLB2 = renteparameter_phi_R_NLA1D100[1].Column1;
13 double renteparameter_Psi_RA2 = renteparameter_Psi_RA1C100[1].Column0;
14 double renteparameter_Psi_RB2 = renteparameter_Psi_RA1C100[1].Column1;
15 double renteparameter_Psi_RC2 = renteparameter_Psi_RA1C100[1].Column2;
16 double voorbeeldRTSE5 = Math.Exp(-1) / (2) * (renteparameter_phi_R_NLB2 + 0.03296211605999014 *
17 (renteparameter_Psi_RA2) + 0.014349731117662046 * (renteparameter_Psi_RB2) + 0.010609776508980583 *
18 (renteparameter_Psi_RC2)) - (1);
19 ... (97 more)
20 double renteparameter_phi_R_NLB100 = renteparameter_phi_R_NLA1D100[99].Column1;
21 double renteparameter_Psi_RA100 = renteparameter_Psi_RA1C100[99].Column0;
22 double renteparameter_Psi_RB100 = renteparameter_Psi_RA1C100[99].Column1;
23 double renteparameter_Psi_RC100 = renteparameter_Psi_RA1C100[99].Column2;
24 double voorbeeldRTSE103 = Math.Exp(-1) / (100) * (renteparameter_phi_R_NLB100 + 0.03296211605999014 *
25 (renteparameter_Psi_RA100) + 0.014349731117662046 * (renteparameter_Psi_RB100) + 0.010609776508980583 *
26 (renteparameter_Psi_RC100)) - (1);
27 double voorbeeldRTSF103 = voorbeeldRTSF102 * (1 + voorbeeldRTSE103);
28 double voorbeeldRTSI1 = voorbeeldRTSF103 - (100000);
29 return voorbeeldRTSI1;
30 }

```

Figure 4.6: A failure in compilation: EXCELERATE failed to detect the chain present in this spreadsheet, resulting in code that is hard to read.

## Idiomaticity

Both versions of the code make use of idiomatic C# principles such as the LINQ calls that enumerate lists instead of using a for-loop. They also utilise the latest language features such as the new collection-expression and the target-typed new, reducing the amount of redundant and repeated class names. Both are recommended by Microsoft as it improves readability by omitting the redundant types [39]. Many variables names adhere to the style guide, with the exception of some of the structures in the new code in Figure 4.4: `TBL_MonthlyExpensesItem` is invalid as it uses underscores and wrong capitalisation. A better name would be `MonthlyExpensesItem`.

This directly complements the next point of critique: the variable names. While the variable names directly link back to the spreadsheet, allowing for traceability of the code back to the spreadsheet, they are not descriptive in any way. This is especially clear in Figure 4.6, where without the context of the spreadsheet, it would be hard to determine the nature of the code. Names of structures can also be improved. The name `WithdrawalCalculatorD26G287` is based on the location of the sheet and range, but it says nothing about the meaning of the structure. This reduces the comprehensibility significantly.

```
ExceLerate > Retirement Planner
1  public class WithdrawalCalculatorD26G287
2  {
3      public double WithdrawalAmountBaseCase { get; set; }
4      public Dictionary<int,double> _withdrawalAmountAtMemoization { get; set; } = new();
5      ...
6
7      public double WithdrawalAmountAt(int counter)
8      {
9          int key = counter;
10         if (_withdrawalAmountAtMemoization.ContainsKey(key))
11         {
12             return _withdrawalAmountAtMemoization[key];
13         }
14
15         if (Equals(counter, 0))
16         {
17             return WithdrawalAmountBaseCase;
18         }
19
20         double result = Math.Min(BalanceAt(counter - (1)) + InterestEarnedAt(counter - (0)), 1d + 0.025d / (12d) *
21         (WithdrawalAmountAt(counter - (1))));
22         _withdrawalAmountAtMemoization.Add(key, result);
23         return result;
24     }
25 }
```

Figure 4.7: A snippet from the *Retirement Planner* showcasing a chain.

A more complex example of a chain can be seen in Figure 4.7. The code does not follow the Microsoft Guidelines in the way the properties of the classes are structured: the guidelines state that `_withdrawalAmountAtMemoization` should be a readonly private field, instead of a public property [39]. This may cause confusion amongst developers and thus reduces comprehension. Furthermore, the use of redundant parenthesis is not beneficial for readability, although this was also present in the code emitted by the old compiler, as can be seen in Figure 4.8.

```
Basic compiler > Monthly Budget
1  double interestD63 = interestJ9 * (interestF62);
2  double interestF63 = interestF62 + interestD63 + 500;
3  double interestD64 = interestJ9 * (interestF63);
4  double interestF64 = interestF63 + interestD64 + 500;
5  double interestD65 = interestJ9 * (interestF64);
6  double interestF65 = interestF64 + interestD65 + 500;
```

Figure 4.8: A snippet of the Monthly Budget workbook compiled with the ‘basic’ compiler. It shows redundant parenthesis on lin 1, 3, and 5.

Finally, in both versions, the structure of the code can be improved by introducing whitespace. Both listings do not use empty lines to separate the structures, which could help structurally separating functionality. Furthermore, there is no use of functions to explicitly separate related functionality. For instance, the interest calculations in Figure 4.9 could have been in a function.

```

Excelerate > Monthly Budget
1  public double Main()
2  {
3      List<MonthlyBudgetReportC14F17Item> monthlyBudgetReportC14F17 = [new(6000, 5800), ..., new(2500, 1500)];
4      double monthlyBudgetReportE9 = monthlyBudgetReportC14F17.Select(t => t.Actual).Sum();
5      InterestC4F65 interestC4F65 = new([500, 500, ..., 500, 500]);
6      double interestF65 = interestC4F65.TotalAt(60);
7      double interestF5 = interestC4F65.TotalAt(0);
8      double interestJ11 = interestC4F65.Deposit.Sum();
9      double interestJ12 = interestF65 - (interestF5) - (interestJ11);
10     List<TBL_MonthlyExpensesItem> tBL_MonthlyExpenses = [new(40, 40), new(0, 0), ..., new(0, 0), new(450, 450)];
11     double monthlyBudgetReportE8 = tBL_MonthlyExpenses.Select(t => t.ActualCost).Sum();
12     double monthlyBudgetReportE7 = monthlyBudgetReportE9 + interestJ12 - (monthlyBudgetReportE8);
13     double monthlyBudgetReportD9 = monthlyBudgetReportC14F17.Select(t => t.Projected).Sum();
14     double monthlyBudgetReportD8 = tBL_MonthlyExpenses.Select(t => t.ProjectedCost).Sum();
15     double monthlyBudgetReportD7 = monthlyBudgetReportD9 + interestJ12 - (monthlyBudgetReportD8);
16     double monthlyBudgetReportF7 = monthlyBudgetReportE7 - (monthlyBudgetReportD7);
17     return monthlyBudgetReportF7;
18 }

```

Figure 4.9: The full *Main* method of the compiled version of the Monthly Budget workbook.

These shortcomings highlight that—while there are positive properties of the code—there is still room for improvement in making the code idiomatic. Based on the above facts, we do note that the code that was produced by EXCELERATE was more idiomatic than the code produced by the ‘basic’ compiler.

### 4.3.2. Threats to validity

Although these experiments show clear semantic equality and improvements in performance, there are several factors that limit the generalisability of our findings.

The semantic equality experiment provided evidence of semantic equality, but it was not proven. We did not provide proof that EXCELERATE produces the exact same semantics due to the scope of the thesis, and instead opted to verify whether the semantic equality was present with a considerable amount of tests. While the results are promising, we cannot say for sure there is semantic equality. This is further threatened by the limited set of spreadsheets that we chose. While semantic equality was preserved between these spreadsheets, we cannot guarantee it works for all spreadsheets and all undiscovered edge cases.

Another threat to construct validity is the COM interop, which adds significance noise and variance to the results of the performance of Excel. It can be seen as fair to include the overhead of the COM interface in the results since Excel engine and EXCELERATE both have to be called from C#. However, we cannot directly compare while we do see differences in speed when the spreadsheet is of a different size, ultimately, we cannot compare this directly with the Excel calculation engine until we know the overhead the COM interface creates.

Readability is subjective, which means that the author with four years of C# experience may have different standards than other, more experienced developers. We try to stick to the literature and the .NET guidelines. However, this subjectivity still introduces a threat to validity.

External validity is threatened by the limited set of simpler spreadsheets. Spreadsheets used in the real world are far more complex. Due to the supported set of features by the compiler, generalisation of the results to real-world spreadsheets is hard.

# 5

## *Chapter 5* **Conclusion**

5.1. Conclusion .....	67
5.2. Future Work .....	68

## 5.1. Conclusion

In this thesis, we set out to create an Excel compiler that produces clear, human-readable C# code using experimental compiler design. We introduce *structure-aware compilation* that uses the structure of the Excel file as heuristic for more readable code. We found two structures commonly used in Excel files and implemented a three-phase compiler called EXCELERATE that utilises structure-aware compilation using a separate IR for each stage.

This study was conducted to answer four research questions:

- (RQ1) How can complex Excel formula compositions be mapped to human-readable C# code?
- (RQ2) What are common excel structures and how can they be applied to optimise the compilation of excel formula compositions?
- (RQ3) What are the performance differences between EXCELERATE (compiled code) and Excel?
- (RQ4) How can the mapping between excel formulas and code be verified?

### 5.1.1. Mapping Excel formulae to human-readable C# code

We employed a multi-phase compiler to transform the complex Excel formula compositions present in the *Worksheet* to human-readable C# code. This multi-phase compiler was built to be modular, and separates the different phases that we had to use. The *Structure Phase* constructs the *Structure Model*: A rich AST of the Excel file, supplemented with *Structures* found within the *Spreadsheets*.

This AST is used in the *Compute Phase* to create the *Compute Grid*, the first part of the *Compute Model*. It allows the compiler to insert the found structures and adjust references to these structures in other nodes. These nodes are linked to form the *Compute Graph*: a representation of the Excel computational model.

The *Compute Graph* is then transformed to the *Code Model* in the *Code Phase* where we apply final transformations to make the code human readable. Finally, we emit the *Code Phase* to C# using the Roslyn Compiler.

### 5.1.2. Common Excel Structures

We identified two structures based on the Microsoft Create Excel Template Repository: the *Table* and the *Chain*. The *Table* models a conventional rectangular worksheet region. The *Chain* is a table with rows that depend on previous rows.

During the *Structure Phase*, we detect these structures and use their information to inject semantics about the domain of the spreadsheet into the generated C# code. Furthermore, the detection of these structures eliminates repetitive, especially in chain structures. EXCELERATE uses separate classes to remove this code duplication from the final generation. As a result, the compiler optimises the code for readability and performance.

### 5.1.3. Verification

We employed randomized differential testing against Excels native calculation engine in order to verify the semantic preservation between Excel and EXCELERATE. The results in Chapter 4 show strong evidence for semantic equality, suggesting the mapping is successful for the supported subset of the Excel functions and operators. Furthermore, a qualitative analysis was done on the readability of the code, resulting in improvements from ‘basic’ compilation without *structure-aware compilation*, providing evidence that the code is human-readable while concluding that improvements can still be made.

### 5.1.4. Performance differences

The evaluation in Chapter 4 answered the final research question by conducting an experiment comparing Excel with compiled EXCELERATE code using randomised inputs. There were no discrepancies found in semantic equality. We observed a linear trend between spreadsheet complexity and calculation time. Furthermore, EXCELERATE was significantly faster, achieving an average speedup of 1710x. The omission of the COM interface played a big role in this. The significant speedup over Excel allows EXCELERATE to substitute in business applications where performance is critical.

Due to scope limits, only a small subset of Excels functions and features was mapped. This limits generalisability of the results to all spreadsheets. Furthermore, due to the COM interface overhead that we have to use during testing, the results for speedup may be skewed if this interface can be removed when communicating with Excel.

Ultimately, this thesis demonstrates that Excels underlying computational model can be compiled into clean, idiomatic C# code with verified semantics and significant performance gains. For companies that are extensively using excel, this can open the door to faster and safer execution.

## 5.2. Future Work

The limitation we described in Chapter 3 directly fuel the future work that can be done on the compiler. In this section, we briefly iterate over possible directions for future work.

An obvious direction is to expand the current supported feature set. This can be as simple as adding formulas, and types like strings we do not support yet. However, more interesting is adding formula's like IF() that would result in conditional logic. This brings challenges like the type inference as we discussed in Section 2.3.2. The most challenging addition would be support for Excel dynamic arrays and formulas that were introduced in Excel 2021. These features allow to work with matrices that 'spill out' into the worksheet, essentially creating cells that have different types of content depending on the calculation or input. Finally, support for a dynamic range operator like INDEX would greatly increase the amount of spreadsheets supported. This operator allows for the range to be dependent on the value of a cell. The last two features would require a rework of the current *Compute Model* to support this dynamic behaviour, since it may require an interpreter.

The compiler can always have more optimisations that increase the readability of the code. We discussed this already in Section 4.3, where we indicated the compiler could benefit from extra steps that increase the logical separation, such as adding whitespace or extracting functions. For this to work, we need reliable heuristics, which may include the current sheet (Cells on different sheets often compute something different, semantically). Furthermore, the detection of extra structures would improve the *structure-aware compilation*, which may be an area for future work. Besides, using context clues for better variable names may be a great improvement: we can use the neighbouring cells to detect for possible labels and use this as the variable name.

A different optimisation for the compiler is to allow the code to be multi-threaded. Excel is able to use multi-threading in large workbooks [57]. When a computation contains two parts that can be calculated in isolation, there is an opportunity to apply multi-threading. Future work could include the identification of these parts and emit them in a way that supports multiple threads.

Finally, the recent rise of LLM models may be utilised to create a different compiler that uses an LLM (agent) to convert the spreadsheet to code. We expect this should result in more idiomatic code that conveys the domain. However, this directly complements the current deterministic method of compilation. Several checks should be utilised to ensure the converted code is semantically equivalent to the Excel file.

# Acknowledgements

First, I would like to thank Info Support for their continuous support, even when things outside of my control confronted me *head on*. Their understanding and patience during this time was truly appreciated. Especially my process supervisor Nikki Thissen. Without their encouragement and help this thesis would not have been written this easily.

More importantly, I thank my supervisors Bjorn Jacobs and Andrés Goens for their support during this thesis. Their questions and constructive feedback helped me sharpen both the technical and academic quality of this thesis.

I would like to thank Ruben Franquinet for his contribution to the name of *EXCELERATE*, without him the name probably would have been a lot less cool.

Finally, I thank Geert Haans, Thijs Boerefijn, Jens Steenmetz, and Ruben Franquinet for helping with proofreading the thesis and being the rubber dummies in my writing process. Their suggestions aided in a significant improvement in clarity.

## Bibliography

- [1] A. Gordon *et al.*, ‘LAMBDA: The ultimate Excel worksheet’. [Online]. Available: [https://www.microsoft.com/en-us/research/blog/lambda-the-ultimate-excel-worksheet-function/?OCID=msr\\_blog\\_lambda\\_tw](https://www.microsoft.com/en-us/research/blog/lambda-the-ultimate-excel-worksheet-function/?OCID=msr_blog_lambda_tw)
- [2] Ministerie van Binnenlandse Zaken en Koninkrijksrelaties, ‘Wetstechnische informatie van Pensioenwet’. Accessed: Sep. 15, 2025. [Online]. Available: <https://wetten.overheid.nl/BWBR0020809/2025-03-01/0/informatie#tab-wijzigingenoverzicht>
- [3] K. Lano *et al.*, ‘Agile Model-driven Engineering of Financial Applications.’, in *MoDELS (Satellite Events)*, 2017, pp. 388–392. Accessed: Jan. 13, 2025. [Online]. Available: <https://nms.kcl.ac.uk/kevin.lano/finmdd/flexmde17f.pdf>
- [4] G. Engels *et al.*, ‘ClassSheets: automatic generation of spreadsheet applications from object-oriented specifications’, in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, in ASE '05. New York, NY, USA: Association for Computing Machinery, Nov. 2005, pp. 124–133. doi: 10.1145/1101908.1101929.
- [5] J. Cunha *et al.*, ‘Spreadsheet Engineering’, *Central European Functional Programming School: 5th Summer School, CEFPS 2013, Cluj-Napoca, Romania, July 8-20, 2013, Revised Selected Papers*. Springer International Publishing, Cham, pp. 246–299, 2015. doi: 10.1007/978-3-319-15940-9\_6.
- [6] J. Cunha *et al.*, ‘Automatically Inferring ClassSheet Models from Spreadsheets’, in *2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, Sep. 2010, pp. 93–100. doi: 10.1109/VLHCC.2010.22.
- [7] J. Cunha *et al.*, ‘From relational ClassSheets to UML+OCL’, in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, Trento Italy: ACM, Mar. 2012, pp. 1151–1158. doi: 10.1145/2245276.2231957.
- [8] P. Sestoft, *A spreadsheet core implementation in C#*, Version 1.0 of 2006-09-28., no. TR-91(2006). in IT University technical report series / IT University of Copenhagen. Copenhagen, Denmark: IT University of Copenhagen, 2006.
- [9] T. S. Iversen, ‘Runtime code generation to speed up spreadsheet computations’, 2006.

- [10] M. Poulsen *et al.*, ‘Optimized Recalculation for Spreadsheets with the Use of Support Graph’, 2007.
- [11] EPPlus Software, ‘EPPlus’. [Online]. Available: <https://github.com/EPPlusSoftware/EPPlus/wiki/Formula-Calculation>
- [12] Aspose, ‘Espose Cells’. [Online]. Available: <https://products.aspose.com/cells/>
- [13] Apache Software, ‘Apache POI’. [Online]. Available: <https://poi.apache.org/components/spreadsheet/eval.html>
- [14] Syncfusion, ‘Syncfusion .NET Excel Library’. [Online]. Available: <https://www.syncfusion.com/document-processing/excel-framework/net/excel-library/formula>
- [15] A. A. Bock *et al.*, ‘On the semantics for spreadsheets with sheet-defined functions’, *Journal of Computer Languages*, vol. 57, p. 100960, Apr. 2020, doi: 10.1016/j.cola.2020.100960.
- [16] P. Sestoft *et al.*, ‘Sheet-Defined Functions: Implementation and Initial Evaluation’, in *End-User Development*, Y. Dittrich, M. Burnett, A. Mørch, and D. Redmiles, Eds., Berlin, Heidelberg: Springer, 2013, pp. 88–103. doi: 10.1007/978-3-642-38706-7\_8.
- [17] D. Steinhöfel *et al.*, ‘Modular, Correct Compilation with Automatic Soundness Proofs’, in *Leveraging Applications of Formal Methods, Verification and Validation. Modeling*, T. Margaria and B. Steffen, Eds., Cham: Springer International Publishing, 2018, pp. 424–447. doi: 10.1007/978-3-030-03418-4\_25.
- [18] G. Rothermel *et al.*, ‘A methodology for testing spreadsheets’, *ACM Transactions on Software Engineering and Methodology*, vol. 10, no. 1, pp. 110–147, Jan. 2001, doi: 10.1145/366378.366385.
- [19] M. Fisher *et al.*, ‘Automated test case generation for spreadsheets’, in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, May 2002, pp. 141–151. doi: 10.1145/581356.581359.
- [20] B. Roziere *et al.*, ‘Unsupervised Translation of Programming Languages’, in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2020, pp. 20601–20611. Accessed: Jan. 29, 2025. [Online]. Available: <https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html>
- [21] R. Waters, ‘Program translation via abstraction and reimplementations’, *IEEE Transactions on Software Engineering*, vol. 14, no. 8, pp. 1207–1228, Aug. 1988, doi: 10.1109/32.7629.
- [22] D. Ordóñez Camacho *et al.*, ‘Automated generation of program translation and verification tools using annotated grammars’, *Science of Computer Programming*, vol. 75, no. 1, pp. 3–20, Jan. 2010, doi: 10.1016/j.scico.2009.10.003.
- [23] J. Cockx *et al.*, ‘Reasonable Agda is correct Haskell: writing verified Haskell using agda2hs’, in *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*, Ljubljana Slovenia: ACM, Sep. 2022, pp. 108–122. doi: 10.1145/3546189.3549920.
- [24] A. Lopes *et al.*, ‘Chomsky: A Content Language Translation Agent’, in *Multi-Agent Systems and Applications IV*, M. Pěchouček, P. Petta, and L. Z. Varga, Eds., Berlin, Heidelberg: Springer, 2005, pp. 535–538. doi: 10.1007/11559221\_54.
- [25] K. Lano *et al.*, ‘Using model-driven engineering to automate software language translation’, *Automated Software Engineering*, vol. 31, no. 1, p. 20, Feb. 2024, doi: 10.1007/s10515-024-00419-y.
- [26] Z. Yang *et al.*, ‘Exploring and Unleashing the Power of Large Language Models in Automated Code Translation’, *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1585–1608, Jul. 2024, doi: 10.1145/3660778.

- [27] X. Chen *et al.*, ‘Tree-to-tree Neural Networks for Program Translation’, in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2018. Accessed: Jan. 29, 2025. [Online]. Available: [https://proceedings.neurips.cc/paper\\_files/paper/2018/hash/d759175de8ea5b1d9a2660e45554894f-Abstract.html](https://proceedings.neurips.cc/paper_files/paper/2018/hash/d759175de8ea5b1d9a2660e45554894f-Abstract.html)
- [28] D. Guo *et al.*, ‘GraphCodeBERT: Pre-training Code Representations with Data Flow’, presented at the International Conference on Learning Representations, Oct. 2020. Accessed: Jan. 29, 2025. [Online]. Available: <https://openreview.net/forum?id=jLoC4ez43PZ>
- [29] D. Lea *et al.*, ‘Idiomatic’. Oxford University Press, Oxford, 2025. [Online]. Available: <https://www.oxfordlearnersdictionaries.com/definition/english/idiomatic>
- [30] JetBrains, ‘Software Developers Statistics 2024 - State of Developer Ecosystem Report’, 2024. Accessed: Sep. 03, 2025. [Online]. Available: <https://www.jetbrains.com/lp/devecosystem-2024>
- [31] J. Börstler *et al.*, ‘“I know it when I see it” Perceptions of Code Quality: ITiCSE '17 Working Group Report’, in *Proceedings of the 2017 ITiCSE Conference on Working Group Reports*, Bologna Italy: ACM, Jan. 2018, pp. 70–85. doi: 10.1145/3174781.3174785.
- [32] S. McConnell, *Code complete*, 2nd ed. Redmond, Wash: Microsoft Press, 2004.
- [33] H. Hunter-Zinck *et al.*, ‘Ten simple rules on writing clean and reliable open-source scientific software’, *PLOS Computational Biology*, vol. 17, no. 11, p. e1009481, Nov. 2021, doi: 10.1371/journal.pcbi.1009481.
- [34] M. Fowler *et al.*, *Refactoring: improving the design of existing code*, Second edition. in The Addison-Wesley signature series. Boston Columbus New York San Francisco Amsterdam Cape Town Dubai London Munich: Addison-Wesley, 2019.
- [35] R. J. Winter, ‘Agile Software Development: Principles, Patterns, and Practices: Robert C. Martin with contributions by James W. Newkirk and Robert S. Koss’, *Performance Improvement*, vol. 53, no. 4, pp. 43–46, Apr. 2014, doi: 10.1002/pfi.21408.
- [36] A. Hunt *et al.*, *The pragmatic programmer: from journeyman to master*, 26. print. Boston: Addison-Wesley, 2011.
- [37] J. Börstler *et al.*, ‘Developers talking about code quality’, *Empirical Software Engineering*, vol. 28, no. 6, p. 128, Nov. 2023, doi: 10.1007/s10664-023-10381-0.
- [38] S. Fakhoury *et al.*, ‘Measuring the impact of lexical and structural inconsistencies on developers’ cognitive load during bug localization’, *Empirical Software Engineering*, vol. 25, no. 3, pp. 2140–2178, May 2020, doi: 10.1007/s10664-019-09751-4.
- [39] Microsoft, ‘.NET Coding Conventions - C#’. Accessed: Sep. 08, 2025. [Online]. Available: <https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions>
- [40] Microsoft, ‘Overview of Excel tables - Microsoft Support’. Accessed: Sep. 19, 2025. [Online]. Available: <https://support.microsoft.com/en-us/office/overview-of-excel-tables-7ab0bb7d-3a9e-4b56-a3c9-6c94334e492c>
- [41] Microsoft, ‘Overview of PivotTables and PivotCharts - Microsoft Support’. Accessed: Sep. 19, 2025. [Online]. Available: <https://support.microsoft.com/en-us/office/overview-of-pivottables-and-pivotcharts-527c8fa3-02c0-445a-a2db-7794676bce96>
- [42] Microsoft, ‘Overview of formulas in Excel - Microsoft Support’. Accessed: Sep. 19, 2025. [Online]. Available: <https://support.microsoft.com/en-us/office/overview-of-formulas-in-excel-ecfdc708-9162-49e8-b993-c311f47ca173>

- [43] Microsoft, 'Structure of a SpreadsheetML document'. Accessed: Sep. 10, 2025. [Online]. Available: <https://learn.microsoft.com/en-us/office/open-xml/spreadsheet/structure-of-a-spreadsheetml-document>
- [44] E. Aivaloglou *et al.*, 'A grammar for spreadsheet formulas evaluated on two large datasets', in *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Bremen, Germany: IEEE, Sep. 2015, pp. 121–130. doi: 10.1109/SCAM.2015.7335408.
- [45] D. Weise *et al.*, 'Value dependence graphs: representation without taxation', in *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '94*, Portland, Oregon, United States: ACM Press, 1994, pp. 297–310. doi: 10.1145/174675.177907.
- [46] Microsoft, 'Floating-point arithmetic may give inaccurate result in Excel - Microsoft 365 Apps'. Accessed: Sep. 11, 2025. [Online]. Available: <https://learn.microsoft.com/en-us/troubleshoot/microsoft-365-apps/excel/floating-point-arithmetic-inaccurate-result>
- [47] 'IEEE Standard for Floating-Point Arithmetic', no. IEEE754–2019. IEEE, Jul. 22, 2019.
- [48] Stack Overflow, 'Stack Overflow 2024 Developer Survey', 2024. [Online]. Available: <https://survey.stackoverflow.co/2024>
- [49] J. Hopcroft *et al.*, 'Efficient algorithms for graph manipulation', *Communications of the ACM*, vol. 16, no. 6, pp. 372–378, Jun. 1973, doi: 10.1145/362248.362272.
- [50] S. Bradner, 'Key words for use in RFCs to Indicate Requirement Levels'. Accessed: Sep. 17, 2025. [Online]. Available: <https://www.ietf.org/rfc/rfc2119.txt>
- [51] O. Kaser *et al.*, 'On the conversion of indirect to direct recursion', *ACM Lett. Program. Lang. Syst.*, vol. 2, no. 1, pp. 151–164, Mar. 1993, doi: 10.1145/176454.176510.
- [52] D. Michie, "'Memo" Functions and Machine Learning', *Nature*, vol. 218, no. 5136, pp. 19–22, Apr. 1968, doi: 10.1038/218019a0.
- [53] C. Lattner *et al.*, 'MLIR: Scaling Compiler Infrastructure for Domain Specific Computation', in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb. 2021, pp. 2–14. doi: 10.1109/CGO51591.2021.9370308.
- [54] W. M. McKeeman, 'Differential Testing for Software', *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.
- [55] Microsoft, 'Stopwatch Class (System.Diagnostics)'. Accessed: Sep. 08, 2025. [Online]. Available: <https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch>
- [56] Microsoft, 'Managed Execution Process - .NET'. Accessed: Sep. 13, 2025. [Online]. Available: <https://learn.microsoft.com/en-us/dotnet/standard/managed-execution-process>
- [57] Microsoft, 'Multithreaded recalculation in Excel'. Accessed: Sep. 19, 2025. [Online]. Available: <https://learn.microsoft.com/en-us/office/client-developer/excel/multithreaded-recalculation-in-excel>

## A. Full code samples

```
Excelerate > Family Budget > Main
1  public double Main()
2  {
3      List<MonthlyBudgetReportC14F17Item> monthlyBudgetReportC14F17 = [new(6000, 5800), ..., new(2500,
1500)];
4      double monthlyBudgetReportE9 = monthlyBudgetReportC14F17.Select(t => t.Actual).Sum();
5      InterestC4F65 interestC4F65 = new([500, 500, ..., 500, 500]);
6      double interestF65 = interestC4F65.TotalAt(60);
7      double interestF5 = interestC4F65.TotalAt(0);
8      double interestJ11 = interestC4F65.Deposit.Sum();
9      double interestJ12 = interestF65 - (interestF5) - (interestJ11);
10     List<TBL_MonthlyExpensesItem> tBL_MonthlyExpenses = [new(40, 40), new(0, 0), ..., new(0, 0),
new(450, 450)];
11     double monthlyBudgetReportE8 = tBL_MonthlyExpenses.Select(t => t.ActualCost).Sum();
12     double monthlyBudgetReportE7 = monthlyBudgetReportE9 + interestJ12 - (monthlyBudgetReportE8);
13     double monthlyBudgetReportD9 = monthlyBudgetReportC14F17.Select(t => t.Projected).Sum();
14     double monthlyBudgetReportD8 = tBL_MonthlyExpenses.Select(t => t.ProjectedCost).Sum();
15     double monthlyBudgetReportD7 = monthlyBudgetReportD9 + interestJ12 - (monthlyBudgetReportD8);
16     double monthlyBudgetReportF7 = monthlyBudgetReportE7 - (monthlyBudgetReportD7);
17     return monthlyBudgetReportF7;
18 }
```

Figure 1.1: Code emitted by EXCELERATE for the Family Budget workbook.

```
Excelerate > Family Budget > Interest Chain
1  public class InterestC4F65
2  {
3      public List<double> Deposit { get; set; }
4      public Dictionary<int,double> _totalAtMemoization { get; set; } = new Dictionary<int,double>();
5
6      public double InterestAt(int counter) => 0.015 / (12) * (TotalAt(counter - (1)));
7      public double TotalAt(int counter)
8      {
9          int key = counter;
10         if (_totalAtMemoization.ContainsKey(key))
11         {
12             return _totalAtMemoization[key];
13         }
14
15         if (counter == 0)
16         {
17             return 10000;
18         }
19
20         double result = TotalAt(counter - (1)) + InterestAt(counter - (0)) + Deposit[counter - (1)];
21         _totalAtMemoization.Add(key, result);
22         return result;
23     }
24
25     public InterestC4F65(List<double> deposit)
26     {
27         Deposit = deposit;
28     }
29 }
```

Figure 1.2: Code emitted by EXCELERATE for the Interest chain in the Family Budget workbook.

```

Basic Compiler > Family Budget > Main
1 public double Main()
2 {
3     List<double> monthlyBudgetReportE9List =
4     [
5         5800,
6         2300,
7         1500
8     ]
9     double monthlyBudgetReportE9 = monthlyBudgetReportE9List.Sum();
10    double interestJ9 = 0.015 / (12);
11    double interestD6 = interestJ9 * (10000);
12    double interestF6 = 10000 + interestD6 + 500;
13    double interestD7 = interestJ9 * (interestF6);
14    double interestF7 = interestF6 + interestD7 + 500;
15    double interestD8 = interestJ9 * (interestF7);
16    double interestF8 = interestF7 + interestD8 + 500;
17    double interestD9 = interestJ9 * (interestF8);
18    double interestF9 = interestF8 + interestD9 + 500;
19    ...
20    double interestD63 = interestJ9 * (interestF62);
21    double interestF63 = interestF62 + interestD63 + 500;
22    double interestD64 = interestJ9 * (interestF63);
23    double interestF64 = interestF63 + interestD64 + 500;
24    double interestD65 = interestJ9 * (interestF64);
25    double interestF65 = interestF64 + interestD65 + 500;
26    List<double> interestJ11List =
27    [
28        0,
29        500,
30        500,
31        ...
32        500,
33        500
34    ];
35    double interestJ11 = interestJ11List.Sum();
36    double interestJ12 = interestF65 - (10000) - (interestJ11);
37    List<double> monthlyBudgetReportE8List =
38    [
39        40,
40        0,
41        ...
42        0,
43        450
44    ];
45    double monthlyBudgetReportE8 = monthlyBudgetReportE8List.Sum();
46    double monthlyBudgetReportE7 = monthlyBudgetReportE9 + interestJ12 - (monthlyBudgetReportE8);
47    List<double> monthlyBudgetReportD9List = [
48        6000,
49        1000,
50        2500
51    ];
52    double monthlyBudgetReportD9 = monthlyBudgetReportD9List.Sum();
53    List<double> monthlyBudgetReportD8List =
54    [
55        40,
56        0,
57        ...
58        0,
59        450
60    ];
61    double monthlyBudgetReportD8 = monthlyBudgetReportD8List.Sum();
62    double monthlyBudgetReportD7 = monthlyBudgetReportD9 + interestJ12 - (monthlyBudgetReportD8);
63    double monthlyBudgetReportF7 = monthlyBudgetReportE7 - (monthlyBudgetReportD7);
64    return monthlyBudgetReportF7;
65 }

```

Figure 1.3: Code emitted by the 'basic' compiler for the Family Budget workbook.

```

Excelerate > Actuarial Example
1   public double Main(List<TableRenteparameter_Psi_RA1C100Item> renteparameter_Psi_RA1C100,
   List<TableRenteparameter_phi_R_NLA1D100Item> renteparameter_phi_R_NLA1D100)
2   {
3       double renteparameter_phi_R_NLB1 = renteparameter_phi_R_NLA1D100[0].Column1;
4       double renteparameter_Psi_RA1 = renteparameter_Psi_RA1C100[0].Column0;
5       double renteparameter_Psi_RB1 = renteparameter_Psi_RA1C100[0].Column1;
6       double renteparameter_Psi_RC1 = renteparameter_Psi_RA1C100[0].Column2;
7       double voorbeeldRTSE4 = Math.Exp(-(1) / (1) * (renteparameter_phi_R_NLB1 + 0.03296211605999014
   * (renteparameter_Psi_RA1) + 0.014349731117662046 * (renteparameter_Psi_RB1) + 0.010609776508980583 *
   (renteparameter_Psi_RC1))) - (1);
8       double voorbeeldRTSF4 = 100000 * (1 + voorbeeldRTSE4);
9       double renteparameter_phi_R_NLB2 = renteparameter_phi_R_NLA1D100[1].Column1;
10      double renteparameter_Psi_RA2 = renteparameter_Psi_RA1C100[1].Column0;
11      double renteparameter_Psi_RB2 = renteparameter_Psi_RA1C100[1].Column1;
12      double renteparameter_Psi_RC2 = renteparameter_Psi_RA1C100[1].Column2;
13      double voorbeeldRTSE5 = Math.Exp(-(1) / (2) * (renteparameter_phi_R_NLB2 + 0.03296211605999014
   * (renteparameter_Psi_RA2) + 0.014349731117662046 * (renteparameter_Psi_RB2) + 0.010609776508980583 *
   (renteparameter_Psi_RC2))) - (1);
14      ...(97 more)
15      double renteparameter_phi_R_NLB100 = renteparameter_phi_R_NLA1D100[99].Column1;
16      double renteparameter_Psi_RA100 = renteparameter_Psi_RA1C100[99].Column0;
17      double renteparameter_Psi_RB100 = renteparameter_Psi_RA1C100[99].Column1;
18      double renteparameter_Psi_RC100 = renteparameter_Psi_RA1C100[99].Column2;
19      double voorbeeldRTSE103 = Math.Exp(-(1) / (100) * (renteparameter_phi_R_NLB100 +
   0.03296211605999014 * (renteparameter_Psi_RA100) + 0.014349731117662046 * (renteparameter_Psi_RB100) +
   0.010609776508980583 * (renteparameter_Psi_RC100))) - (1);
20      double voorbeeldRTSF103 = voorbeeldRTSF102 * (1 + voorbeeldRTSE103);
21      double voorbeeldRTSI1 = voorbeeldRTSF103 - (100000);
22      return voorbeeldRTSI1;
23  }
24 }

```

Figure 1.4: Code emitted by EXCELERATE for the Actuarial Example workbook.

```

Excelerate > Withdrawal Calculator Chain
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace ExcelCompiler.Generated;
8 public class WithdrawalCalculatorD266287
9 {
10     public List<double> AdditionalWithdrawal { get; set; }
11     public double WithdrawalAmountBaseCase { get; set; }
12     public double BalanceBaseCase { get; set; }
13     public Dictionary<int,double> _withdrawalAmountAtMemoization { get; set; } = new();
14     public Dictionary<int,double> _balanceAtMemoization { get; set; } = new();
15
16     public double InterestEarnedAt(int counter) => BalanceAt(counter - (1)) -
(WithdrawalAmountAt(counter - (1))) * (0.04d / (12d));
17     public double WithdrawalAmountAt(int counter)
18     {
19         int key = counter;
20         if (_withdrawalAmountAtMemoization.ContainsKey(key))
21         {
22             return _withdrawalAmountAtMemoization[key];
23         }
24
25         if (Equals(counter, 0))
26         {
27             return WithdrawalAmountBaseCase;
28         }
29
30         double result = Math.Min(BalanceAt(counter - (1)) + InterestEarnedAt(counter - (0)), 1d +
0.025d / (12d) * (WithdrawalAmountAt(counter - (1))));
31         _withdrawalAmountAtMemoization.Add(key, result);
32         return result;
33     }
34
35     public double BalanceAt(int counter)
36     {
37         int key = counter;
38         if (_balanceAtMemoization.ContainsKey(key))
39         {
40             return _balanceAtMemoization[key];
41         }
42
43         if (Equals(counter, 0))
44         {
45             return BalanceBaseCase;
46         }
47
48         double result = BalanceAt(counter - (1)) - (WithdrawalAmountAt(counter - (0))) -
(AdditionalWithdrawal[counter - (1)]) + InterestEarnedAt(counter - (0));
49         _balanceAtMemoization.Add(key, result);
50         return result;
51     }
52
53     public WithdrawalCalculatorD266287(List<double> additionalWithdrawal, double
withdrawalAmountBaseCase, double balanceBaseCase)
54     {
55         AdditionalWithdrawal = additionalWithdrawal;
56         WithdrawalAmountBaseCase = withdrawalAmountBaseCase;
57         BalanceBaseCase = balanceBaseCase;
58     }
59 }

```

Figure 1.5: Code emitted by EXCELERATE for the Retirement Planner workbook.