

UNIVERSITY OF BERGEN  
DEPARTMENT OF INFORMATICS

---

# Integrating multi-way data-flow constraint systems in spreadsheets

---

*Author:* Torjus Schaathun

*Supervisors:* Mikhail Barash, Jaakko Järvi



UNIVERSITETET I BERGEN  
*Det matematisk-naturvitenskapelige fakultet*

June, 2022

## **Abstract**

Graphical user interfaces (GUIs) can be found in almost all aspects of our lives. We take a particular interest in spreadsheets in thesis, as they are an essential tool in many professions. Microsoft Excel is a prevalent spreadsheet software widely used across industries; it is frequently used to fill out various forms. However, maintaining dependencies between variables can be difficult to ensure and prone to bugs, especially as the size of the project increases. Our approach is to introduce some structure to Excel, and we have thus created a tool, HDTables, to address the problem of maintaining dependencies between variables. HDTables is an Excel add-in created using HotDrink, a JavaScript library that uses multi-way data-flow constraint systems to model GUIs, in combination with the visual programming environment Blockly. The tool is aimed at end-users with no programming background and allows them to create variables and constraints to explicitly define the data-flow between cells in Excel.

## **Acknowledgements**

I would like to thank my two supervisors, Mikhail Barash and Jaakko Järvi, for pushing on having weekly meetings and having my back throughout the thesis. I would not have made it without such excellent supervisors.

I would also like to thank my parents for bringing me food supplies over the years and encouraging and motivating me.

Lastly, I would like to thank Amund, Henrik, Kristian, and Simen (Brofavors).

**Torjus Schaathun**

Wednesday 1<sup>st</sup> June, 2022

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Thesis Outline . . . . .	2
<b>2</b>	<b>Background</b>	<b>3</b>
2.1	Constraint Systems . . . . .	4
2.2	Block-based Specifications . . . . .	7
2.3	Empirical Evidence For and Against Visual programming Languages . . .	8
2.4	Spreadsheets . . . . .	9
<b>3</b>	<b>Constaint systems in spreadsheets</b>	<b>17</b>
3.1	Excel Add-ins . . . . .	18
3.2	HDTables . . . . .	18
<b>4</b>	<b>Implementation</b>	<b>24</b>
4.1	Architecture . . . . .	24
4.2	Add-in implementation . . . . .	25
4.3	Blockly implementation . . . . .	26
4.4	Code generation . . . . .	28
<b>5</b>	<b>User study</b>	<b>31</b>
5.1	Experienced Programmers: Results . . . . .	32
5.2	Non-programmers: Results . . . . .	33
5.3	Feedback . . . . .	34
5.4	Summary . . . . .	34
<b>6</b>	<b>Related work</b>	<b>35</b>
6.1	XLBlocks . . . . .	35
6.2	Block Shelves . . . . .	37
6.3	Polaris . . . . .	38
6.4	TrueGrid . . . . .	39

6.5	ZenSheet Studio . . . . .	40
6.6	Spreadsheet Functional Programming . . . . .	40
6.7	Gradual Structuring . . . . .	41
6.8	Object Oriented Functional Spreadsheets . . . . .	41
6.9	SpreadsheetDoc . . . . .	42
6.10	ClassSheets . . . . .	42
6.11	Tabula . . . . .	42
<b>7</b>	<b>Discussion and Future Work</b>	<b>44</b>
	<b>Bibliography</b>	<b>47</b>
<b>A</b>	<b>Code generated from Blockly specifications</b>	<b>52</b>

# List of Figures

2.1	Example of the GUI for converting US Dollar to Norwegian Krone in Google.	3
2.2	Example of Blockly code.	7
2.3	Spreadsheet example showing formulas.	10
2.4	Graph-oriented view of Figure 2.3.	11
2.5	Dependencies between cells from Figure 2.3.	12
2.6	Income and tax example with dependencies in Excel.	14
2.7	Income and tax example with dependencies in Excel.	14
2.8	Income and tax example with dependencies in Excel.	15
2.9	How the dependencies in Figure 2.8 should be.	16
3.1	Our add-in running in Excel.	19
3.2	Blockly code for the example in Figure 3.1.	20
3.3	Example of the method block inside the Method category.	21
3.4	Example of the contents in the Variables category.	22
3.5	Example of a variable block.	22
4.1	Architecture of our prototype implementation.	25
5.1	Performance on each task for experienced programmers	32
5.2	Performance on each task for non-programmers	33
6.1	Example of a block definition of a SUMIFS formula [23]	36
6.2	Example of a formula that makes reference to multiple worksheets [22].	38
6.3	Example of a TrueGrid implementation, with the code editor on the left and the grid on the right side [20].	39

# List of Tables

4.1	Correspondence between our tool, HotDrink code, and JavaScript/Hot-Drink API. . . . .	30
A.1	Correspondence between visual Blockly specifications in our tool, Hot-Drink code, and JavaScript/HotDrink API. . . . .	52

# Listings

2.1	Example of a constraint between Fahrenheit and Celsius using HotDrink.	4
4.1	Code for adding an event handler on the selected cell. . . . .	26
4.2	Code to setup and assemble the code from a block. . . . .	27
4.3	Code to setup a toolbox in Blockly . . . . .	27
4.4	JSON returned from method block . . . . .	28
4.5	JSON returned from method block . . . . .	29



# Chapter 1

## Introduction

There are many different types of user interfaces in the world, and many of them are used to fill out forms. We are particularly interested in spreadsheets because they are used as an essential tool in many professions, primarily to fill out forms. Microsoft Excel is one of the most prevalent spreadsheet technologies [38], frequently used to create tables and fill out forms. However, spreadsheets are generally prone to bugs and errors. Excel-integrated Visual Basic for Applications and other approaches are often layered on top of Excel spreadsheets to validate the correctness of the entered data values and improve users' experience with such forms within spreadsheets. Even with these additional validation layers, spreadsheets are still prone to errors.

We want to introduce some structure to Excel and allow Excel users to explicitly define the data-flow between cells, which is impossible in the traditional Excel model. We present a tool that introduces multi-way data-flow constraint systems to spreadsheets. Our tool, HDTables, is a Microsoft Excel add-in that allows users to specify the data-flow between cells in Excel explicitly. HotDrink is a JavaScript library for creating multi-way data-flow constraint systems that allow for the explicit definition of data-flow between cells in Excel. However, we cannot assume that Excel users have any knowledge of JavaScript, and we address this problem by implementing the visual programming environment Blockly. Blockly and other block-based environments let users write code using blocks. Blockly blocks are shaped like jigsaw puzzle pieces in various colors, making it easy for people with little or no programming experience to understand how the pieces connect. It is also nearly impossible to make syntactic errors when using blocks, which is advantageous when used by a non-programmer. Participants in our user study expressed this sentiment, and our prototype implementation demonstrates the feasibility of directly specifying multi-way data-flow constraint systems in spreadsheets for non-developer users.

## 1.1 Thesis Outline

**Chapter 1** gives an introduction and overview of the problem, and how we solve it.

In **Chapter 2** we present an overview of how multi-way data-flow constraint systems works, we introduce a block-based visual programming language, and we explain spreadsheets.

**Chapter 3** continues with an overview of how multi-way data-flow constraint systems works with spreadsheets, and shows an example of our prototype. This should help the reader understand how the prototype is used, and what its usages are.

**Chapter 4** discusses the implementation of multi-way data-flow constraint systems in spreadsheets.

In **Chapter 5** we discuss how users utilized our solution, how they behaved, their needs and their feedback.

**Chapter 6** presents related work, why it is relevant and what is similar.

**Chapter 7** gives a discussion and summary of our work.

# Chapter 2

## Background

Graphical user interfaces (GUIs) are ubiquitous, appearing on our computers, phones, tablets, train stations, and so on. However, GUIs tend to be buggy sometimes, which creates problems for the user, particularly when there are non-trivial dependencies between values that the user is supposed to edit. A simple example of such dependencies is shown in Figure 2.1. When searching for the conversion from US Dollar to Norwegian Krone in Google, Google presents a GUI for converting between them, where changing one of the fields updates the other. This requires two methods to update the values of the variables in a mutually dependent manner.

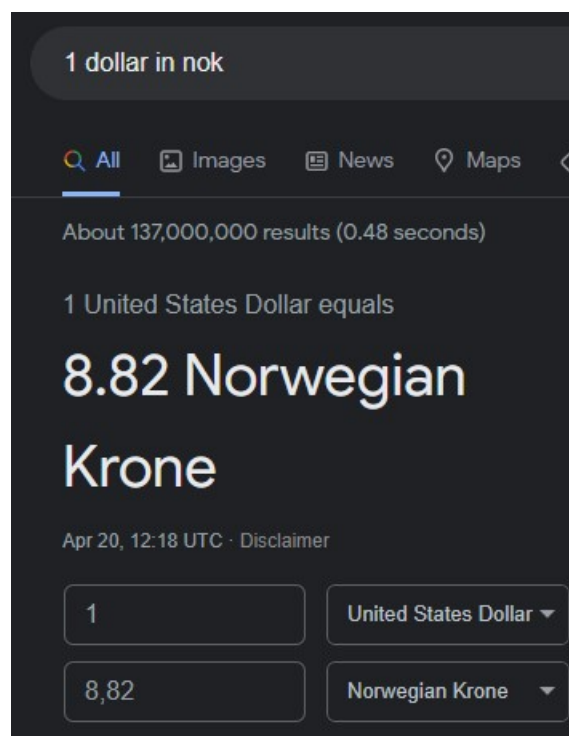


Figure 2.1: Example of the GUI for converting US Dollar to Norwegian Krone in Google.

## 2.1 Constraint Systems

A possible solution to fix buggy graphical user interfaces (GUIs) is to introduce constraint systems. A constraint system [24] can be presented as a tuple  $\langle V, C \rangle$ , where  $V$  is a set of variables, and  $C$  is a set of constraints. Every constraint in the set is a tuple  $\langle R, r, M \rangle$ , with  $R \subseteq V$ . The  $r$  represents some  $n$ -ary relation between the variables in  $R$ , with  $n = |R|$ , and  $M$  is a set of *constraint satisfaction methods*. A constraint is satisfied if the values of variables in  $R$  satisfy  $r$ .

We use HotDrink [18] in this thesis to model constraint systems. HotDrink is a JavaScript library that utilizes multiway data-flow constraint systems in order to model GUIs. As it is a JavaScript library, it works especially well for web-based GUIs. A simple example of a constraint system is an application to convert temperature between Fahrenheit and Celsius, where the value of Fahrenheit is dependent on the value of Celsius and vice versa. Listing 2.1 below shows the HotDrink code for a constraint system with Fahrenheit and Celsius.

Listing 2.1: Example of a constraint between Fahrenheit and Celsius using HotDrink.

```
1 component comp {
2   var c = 1, f;
3   constraint {
4     (f -> c) => (f - 32) / 1.8;
5     (c -> f) => (c * 1.8) + 32;
6   }
7 }
```

When using HotDrink, it is the programmer's responsibility to create and handle bindings between the model and the view-model. As the classic Model-View-Controller pattern [26], the model is the single authority on the application's logic and "business" data. The HotDrink API essentially provides an embedded domain-specific language for defining view-models, recognized as multi-way data-flow constraint systems. HotDrink lets programmers declare variables and relationships between them, which comprise the constraint system in the view-model.

A *variable* in HotDrink can be written by calling it as a function with the new value as an argument, and read by calling it with no arguments. *Constraints* in HotDrink describe the relationship between variables. Each constraint in the constraint system consists of *methods*. A method establishes the relationship between variables by computing new values for the variables in the constraint system, thus satisfying the constraint.

The constraints and variables form a multi-way data-flow constraint system. Executing one method in each constraint in an order that does not cause already satisfied constraints to be invalidated solves the constraint system.

The view-model updates when adding a new variable or constraint, writing to a variable, or calling `system.update()`. Updating sends out notifications of each variable change and solves the constraint system. Event handlers are necessary for the constraint system to communicate with and update the view. The event handlers handle the bindings from the view-model to the model and from the view-model to the view.

To create the constraint from Listing 2.1 using the HotDrink framework, we first start by defining an empty constraint system:

```
const system = defaultConstraintSystem;
```

Then we create an empty component and give it a name:

```
const comp = new Component("MyComponent");
```

Then the component is added to the constraint system:

```
system.addComponent(comp);
```

We create the variables, add them to the component with `null` as the initial value, and bind them to the view:

```
comp.emplaceVariable('f', null);  
comp.emplaceVariable('c', null);  
binder(comp.vs['f'], 'f');  
binder(comp.vs['c'], 'c');
```

The `binder()` function takes a HotDrink variable and a reference to a component in the view as arguments and creates event handlers between them, thus ensuring updates when either variable changes.

We create one method to convert Fahrenheit to Celsius and one method to convert Celsius to Fahrenheit:

```

const toFahrMethod = new Method(2, [0], [1], [maskNone], c =>
  { c * 1.8 + 32 });
const toCelMethod = new Method(2, [1], [0], [maskNone], f =>
  { (f - 32) / 1.8 });

```

The `Method()` object takes as argument the number of variables in the method, the positions of the input variables, the positions of the output variables, a `MaskType`<sup>1</sup>, and a function. We get the positions of the input and output variables from a list containing all variables for the constraint.

We create specifications for the constraint where we add the two methods we created:

```

const cspec = new ConstraintSpec(
  Array.from([toFahrMethod, toCelMethod]));

```

The `ConstraintSpec()` takes an array of methods as an argument and creates the specifications for the constraint.

We refer to both variables from the component, and the reference can be attained in the following way:

```

const fRef = comp.getVariableReference('f');
const cRef = comp.getVariableReference('c');

```

Finally, we are ready to create the constraint by adding the constraint specifications and variable references to the component and updating the constraint system to ensure that the constraint has been set:

```

comp.emplaceConstraint(constraintId, cspec, [cRef, fRef], false);
system.update();

```

---

<sup>1</sup><https://git.app.uib.no/Jaakko.Jarvi/hd4/-/blob/feature/scripting/src/constraint-system/constraint-system-util.js#L67>

## 2.2 Block-based Specifications

To specify a constraint system in HotDrink requires some programming knowledge in JavaScript. However, this cannot be expected by the majority of spreadsheet users. Therefore, we decided to implement a block-based visual programming language (VPL) to let users specify constraint systems without prior programming knowledge.

Block-based environments are generally easy to use. They come with "jigsaw pieces" in different shapes and colors which makes it intuitive for the user to understand how pieces connect. Block-based environments also eliminates the possibility for syntactic errors, which is useful when utilized by an inexperienced programmer.

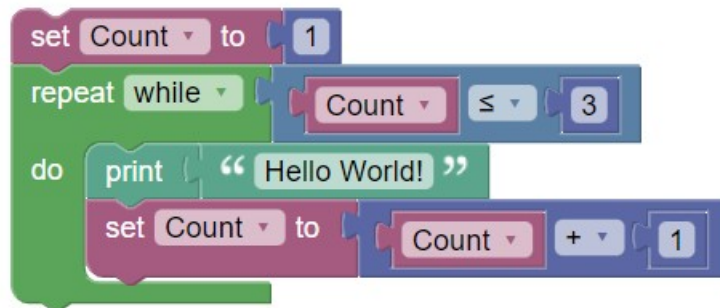


Figure 2.2: Example of Blockly code.

Block-based environments comes predominantly in the two types, *languages* and *tools*. The former refers to programming languages that are written using blocks, in contrast to traditional text-based programming languages. The latter refer to tools that help in the development of block-based environments. Blockly [1] is such a tool, and is what has been used in this thesis.

Blockly is an open source, pure JavaScript library, that is easily extensible and customizable. Blockly code can be exported to many programming languages, such as JavaScript, Python and Dart. At its simplest form the Blockly editor includes a workspace for organizing blocks and a toolbox to store different block types to drag into the workspace. It is also possible to create custom blocks in the application for users to use in the workspace.

## 2.3 Empirical Evidence For and Against Visual programming Languages

This paper by Whitley [43] was published in 1996, and we will look at this paper's proposed evidences for and against VPLs, even though new evidence has emerged since then [30].

Many VPLs and visualization systems have been developed since the 80s as a result of an active visual programming community. There is, however, as of 1996, relatively little empirical evidence supporting the design choices of these systems.

The question this paper asks is "What data exists that tells us when and how visual notations can be beneficially used within the context of the complex cognitive activities required by programming?" [43]. The paper does not focus as much on visualization systems as it does on VPLs. The paper uses the term *visuals* when referring to visual notations (such as geometric shapes, lines and patterns). Thus, we will do the same.

**Evidence Against VPLs** The studies relating to evidences against VPLs are mostly speculation and some have premature conclusions. However, a study by Ramsey et al. [36] brings up the problems of the Deutch Limit, which refers to how visual notations takes up more screen space in comparison to textual notations. The visual programming community have recognized this problem for a long time, a problem that fuels the impracticality of VPLs.

An other study by Green and Petre [34] speculates that secondary notation, which is an issue in textual notations as well as visual notations, will be harder for novice VPL users to master.

**Evidence For VPLs** Organised and consistent information makes it easier to digest the information, especially for people working in problem solving and design. Moreover, explicit information tends to yield more efficient representations of that information. This does not apply only to visuals, but to text as well. Visual notations, compared to textual, can in many cases produce more explicit information and ensure better organization. Furthermore, using visuals provides measurable performance enhancements, such as time and correctness.



In text-based programming, where the problems tend to be larger and more complex than those in controlled experiments, there might exist an important role for VPLs. This was observed in these four studies from the paper: An editing study by Day [15], a study by Polich and Schwartz [35], a study by McGuinness [28] and a flowchart study by Scanlan [39]. Visuals was also observed yielding better performances in smaller problems. This was noted by these studies: the editing study by Day, the flowchart study by Scanlan, a flowchart study by Cunniff and Taylor [13] and a study by Pandey and Burnett [32]. These all show that VPLs could play a role in end-user programming where the problems tend to be smaller and less complex than in traditional text-based programming.

Having looked at these evidences for and against VPLs we can also remark that VPLs have become much more used since 1996 with popular VPLs such as Scratch [8], Blockly [1] and many others, enforcing that VPLs works and makes it easier, for at least some, to program certain things.

We see that the evidences for VPLs appears to outweigh the evidences against, especially when taking into account that these evidences came from a paper published in 1996, and we see that VPLs have become more popular since then. However, the remark of the Deutch Limit may still be a problem in VPLs today, and something that might impact our implementation.

## 2.4 Spreadsheets

This section will follow the structure of the *What is a spreadsheet* chapter in the book *Spreadsheet Implementation Technology: Basics and Extensions* by Sestoft [40].

In 1979, Bricklin and Frankston developed the first spreadsheet program, VisiCalc, for the Apple II computer [14]. Following VisiCalc came various spreadsheets, including Lotus 1-2-3, SuperCalc, QuattroPro, and PlanPerfect. Today's dominating spreadsheet program is Microsoft Excel [3]. Many open-source spreadsheet programs exist, including OpenOffice Calc [7] and Gnumeric [5].

Every spreadsheet program consists of a two-dimensional grid of cells. Rows in the spreadsheet are labeled with numbers 1, 2, ..., columns are labeled with the letters A, B, ..., Z, AA, AB, ..., cells are addressed by row and column: A1, A2, ..., B1, B2, ..., and the area of a selection of cells by their corner coordinates, such as A1:C5. A cell can contain a text, a number, or a formula. A formula can involve functions like SUM(...),

arithmetic operators (such as +, -, \*, /), constants, and references to other cells such as B4, or cell areas such as A5:D9. If the contents of a cell change, the cells that are directly or transitively dependent on that cell will be recalculated.

Another essential feature that modern spreadsheets have in common is that a reference in a formula can be relative (like B3), absolute (like \$D\$4), or a mixture of both, column-relative but row-absolute (like D\$5) or column-absolute but row-relative (like \$D5).

Instead of viewing spreadsheets as rectangular grids of cells, they can be represented as graphs with nodes representing cells and edges representing relationships between cells, as shown in Figure 2.4.

Cell references and cell area references are usually entered and displayed in the format consisting of a row and a column representation, called the *A1 format*, introduced originally by VisiCalc [14]. References are by default relative, and the dollar (\$) prefix indicates an absolute row or column.

	A	B	C	D	E
1		Count	Weight	Total weigth	Count %
2	Water melon	5	3	=B2*C2	=B2/\$B\$5*100
3	Orange	4	0.3	=B3*C3	=B3/\$B\$5*100
4	Apple	7	0.2	=B4*C4	=B4/\$B\$5*100
5	Sum	=SUM(B2:B4)		=SUM(D2:D4)	
6					
7		Average	=D5/B5		

Figure 2.3: Spreadsheet example showing formulas.

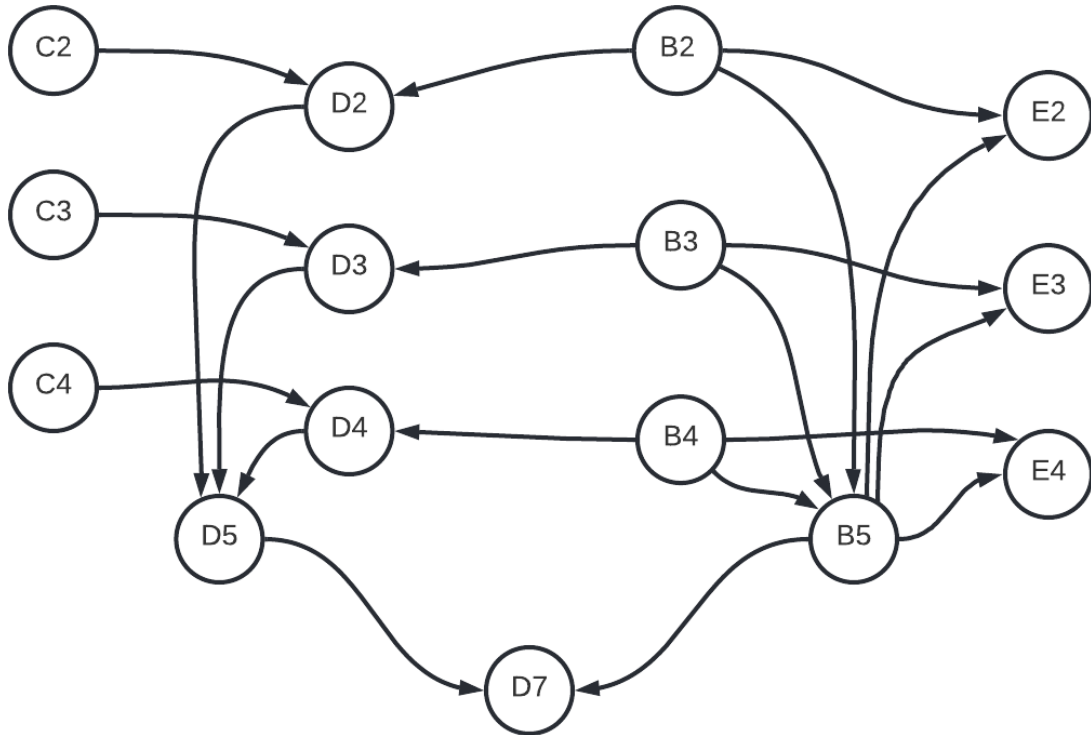


Figure 2.4: Graph-oriented view of Figure 2.3.

As mentioned, a formula in a cell is an expression that can contain standard operators (such as  $+$ ,  $-$ ,  $*$ ,  $/$ ), calls to functions (such as `SUM`), and references to other cells. Most spreadsheet programs contain basic functions such as `LOG`, `EXP`, `SIN`, `RAND`, `IF`, and many more. Some functions take arguments that can be a cell area reference or range, such as `A1:A3`, representing the `A1`, `A2`, and `A3` cells. An area reference generally consists of two cell references (here `A2` and `A4`), giving two corners of a rectangular area of a sheet. The reference to the two corner cells can be any combination of absolute and relative.

Modern spreadsheet programs let users define multiple related sheets in a workbook. Optionally, a cell reference can refer to a cell on another sheet in the same workbook using the notation `Sheet3!A1` in Excel.

A significant part of spreadsheets is the dependencies between cells. We say that cell `D3` *directly depends* on cells `B3` and `C3` when it contains the formula `=B3*C3`, such as in Figure 2.3. We also say that cells `B3` and `C3` *directly support* cell `D3`. Excel supports a feature called *Trace Dependents* that displays the dependencies of a cell. Figure 2.5 shows the traced dependencies of the cells from Figure 2.3.

	A	B	C	D	E
1		Count	Weight	Total weight	Count %
2	Water melon	5	3	15	31.25
3	Orange	4	0.3	1.2	25
4	Apple	7	0.2	1.4	43.75
5	Sum	16		17.6	
6					
7		Average	1.1		

Figure 2.5: Dependencies between cells from Figure 2.3.

All cells directly supported by a cell must be recalculated when that cell changes value, regardless of whether they are in the same sheet in the workbook. Recalculations do not have to happen too often when humans edit the cells. However, some numerical routines may update the spreadsheet more frequently. The order of recalculation is essential. If cell A1 depends on cell A2, then the value of A2 should be calculated before the value A1. Recalculations can start in either top-down order or bottom-up order. Recalculations can start with any cell using *top-down* order. A cell is only computed when its value is needed, and then the original cell can be computed. Recalculations start with cells that do not depend on other cells in bottom-up order, proceeding with cells that only depend on already calculated cells.

The recalculation mechanism of a spreadsheet implementation should be designed to ensure reliability and efficiency. Following is a list of requirements for a recalculation after one cell has been edited:

- Recalculations should be correct. The contents of all cells should be consistent with each other after a recalculation.
- Recalculations should be efficient in time and space.
- Recalculation should detect dynamic cycles accurately. A dynamic cycle happens when a cell dynamically transitively depends on itself.
- Recalculation should not evaluate unused arguments of non-strict functions such as  $\text{IF}(e_1; e_2; e_3)$  and should evaluate volatile functions such as  $\text{RAND}()$  and  $\text{NOW}()$ .

Most spreadsheet programs' built-in functions are strict, meaning that all their arguments are evaluated before they are called. Non-strict functions are functions such as  $\text{IF}(e_1; e_2; e_3)$ , where only one of  $e_2$  or  $e_3$  will be evaluated. Moreover, functions such as  $\text{NOW}()$  and  $\text{RAND}()$  are volatile because, even though they take no argument, they

typically return different values each time they are called. `RAND()` returns a random number, and `NOW()` returns the current time. Volatile functions can complicate the control of recalculation order.



Non-strict functions implicate the presence or absence of cycles. Putting the formula `IF(A1>0; A2; 1)` inside cell `A1` seems to introduce a cyclic dependence of `A2` to `A2`; however, that only happens if cell `A1` evaluates to a positive number, as only arguments that get evaluated can introduce a cycle. Excel works like this by only reporting a cyclic dependency if it has to be evaluated.

Spreadsheet programs dynamically differentiate between many types of data, such as text strings, numbers, arrays, and logical values. It is possible to have cell `B1` contain the function `=IF(C1<0; 0; B1)` as long as `C1<0` is not false; otherwise, the evaluation will contain a cyclic dependency.

A function may fail by giving it the wrong type of argument or the wrong number of arguments. However, such failures are likely to arise when editing a spreadsheet. The spreadsheet should tolerate such failures, and instead of throwing an exception, it should produce an error value, and further computations must propagate the error value. For example, applying the function to raise a number to a power on a string as in `POWER("two", 3)` should give an `ArgType` error value, and further computations must propagate the error value so it can easily be traced back.

Changing any value in a spreadsheet without breaking any dependencies could be desirable. Excel includes the option *Enable iterative calculation* to allow circular dependencies. However, this option does not allow users to change the value of cells with circular dependencies without replacing the formula with the new value. One would need to make multiple copies of a table to achieve the ability to change values without breaking dependencies.

Consider now an example of how a simple calculator of income taxation can be implemented in a spreadsheet. The Norwegian Tax Administration—Skatteetaten—has a page where individuals can calculate how much tax they will owe based on their income, as shown in Figures 2.6 and 2.7. However, as shown in the figures, it is impossible to change any other field than the income field. It can be inconvenient to only be able to change the income field because people may want to know their income if they set the net income or how other fields may change based on other fields.

**Salary and payments in kind, etc.**  

600 000

**Standard minimum deduction**

109 950

Figure 2.6: Income and tax example with dependencies in Excel.

## Your calculation: 157 598 kr.

[Close](#)

Tax	Basis	Amount
Wealth tax state	0	0
Wealth tax municipal	0	0
Income tax municipal	490 050	47 282
Income tax county	490 050	10 363
Common tax	490 050	37 351
Bracket tax	600 000	14 602
National Insurance contributions	600 000	48 000
Sum deductions	0	0

Figure 2.7: Income and tax example with dependencies in Excel.

Figure 2.8 shows four income and tax examples with dependencies traced, where only the yellow value in each example can be changed without breaking any dependencies. We have arranged cells in a zig-zag manner to see the dependencies easily. In the top left example, only income and deduction can be changed, and percentage, tax, and net income are dependent on income, and deduction is dependent on time. The top right example is similar, but the time is dependent on the deduction instead. In the bottom right example, net income and time can be changed, and income is dependent on net income, and deduction is dependent on time. The bottom right is similar, but the time is dependent on the deduction.

	A	B	C	D	E	F	G
1	<b>Income</b>	600000			<b>Income</b>	500000	
2	percent		30		percent		30
3	<b>time</b>	12			time	8	
4	deduction		109956		<b>deduction</b>	75000	
5	tax	147013			tax	75000	
6	net income		452987		net income		425000
7							
8							
9	Income	600000			Income	609410	
10	percent		30		percent		30
11	<b>time</b>	12			time	9	
12	deduction		109956		<b>deduction</b>	88000	
13	tax	147013			tax	156423	
14	<b>net income</b>		452987		<b>net income</b>		452987

Figure 2.8: Income and tax example with dependencies in Excel.

The tables in Figure 2.8 should optimally be fused to only one table, where it is possible to change any value without overwriting the formulas. Figure 2.9 shows how the dependencies may look if they are fused. However, this is not possible with Excel because updating the cells with the derived values will overwrite the formulas in them. Also, as Figure 2.9 may suggest, updating time should not cause an update in income. Income should only be updated when tax or net income directly gets updated. Constraint systems can ensure that the correct variables are updated by only executing one method in each constraint, as described in Section 2.1.

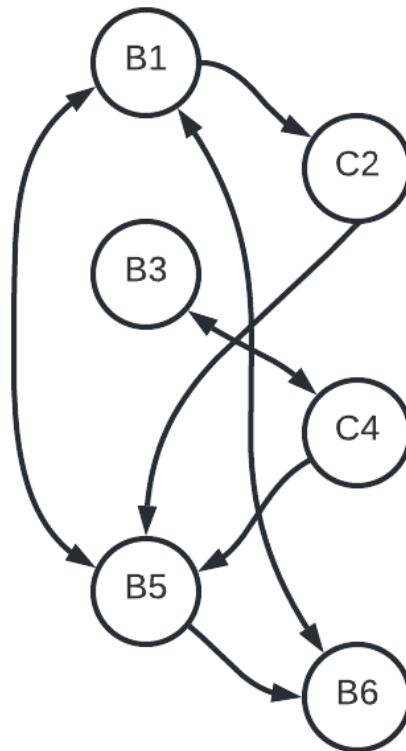


Figure 2.9: How the dependencies in Figure 2.8 should be.

Apart from the features discussed above, most modern spreadsheet programs supply business graphics such as pie charts, bar charts, and scatterplots. They also provide spell checkers, pivot tables, and many other unique and practical features. This is beyond the scope of our thesis.



# Chapter 3

## Constraint systems in spreadsheets

In the previous section, we explained where spreadsheet applications today are limited: expressing spreadsheets where dependencies between cells are “multi-way”. In other words, those are the dependencies that are arbitrary relations, not merely functional ones. The goal of this thesis is to inspect the applicability of multi-way constraint systems as an underlying computational engine underneath a spreadsheet. The conjecture is that multi-way dependencies (like in the taxation example in the previous chapter) are easy and clear to express in such spreadsheets. We conjecture that with such spreadsheets, certain types of mistakes and errors in spreadsheet programs could be less frequent. Before explaining how our system works, we discuss how serious a problem spreadsheet errors are, and then describe the concept of Excel Add-Ons, which is how we have implemented our system.

Errors in spreadsheets are very costly and, unfortunately, not that uncommon. The European Spreadsheet Risk Interest Group (EuSpRIG) [33] has collected and sorted a list of spreadsheet errors that demonstrate common problems that occur with the uncontrolled use of spreadsheets. A recent example of a costly error happened in 2020 when close to 16000 Covid-19 cases went unreported in England [25]. This happened because of how Public Health England (PHE) gathered logs of swab tests from commercial firms in order to find out who had the virus. An automated process to pull the gathered data into Excel was made by PHE, so that the data could be uploaded to a central system. However, the developers from PHE chose XLS as the file format to store the rows, an old file format used primarily between 1997 and 2003 [9]. This format allows for only 65 000 rows each template, instead of the over 1 million rows actually allowed in Excel. Thus, because each test result consists of several rows of data, each template could only hold around 1 400 cases. Ultimately PHE ended up breaking down the test results in smaller batches to avoid the problem, but it should never have been a problem in the first place.

It can be easy to think that experts are so experienced with Excel that they never make mistakes. A famous paper, Growth in Time of Debt [37], by Reinhart and Rogoff, seemed to prove that the growth of a country was approximately cut in half when its external debt reached 90% of the GDP. The paper had a considerable influence, such as people being fired to lower a country's debt and supranational institutions and governments using the paper to support government fell and austerity policies. The problem was that the paper's results were not accurate. Reinhart and Rogoff had forgotten to include an entire column in their calculations. [41]

### 3.1 Excel Add-ins

An Excel add-in [4] gives us the ability to extend the functionality of an Excel application, in multiple platforms, such as Mac, Windows and your browser. Office JavaScript APIs are provided by the Office Add-ins platform. This lets us use standard web technologies such as JavaScript, HTML and CSS, which lets us easily integrate it with HotDrink, as HotDrink is a JavaScript library.

The Office JavaScript API enables our Excel add-in to interact with objects in Excel. The API lets us access cells, write and read data to them, and perform complex functions using their values. Listing 4.1 shows how to retrieve information from the currently selected cell in Excel using the `getActiveCell()` method.

### 3.2 HDTables

As our title suggests, our prototype aims to integrate multi-way data-flow constraint systems into spreadsheets. We have therefore created an Excel add-in and integrated it with HotDrink functionality. We are targeting our implementation at end-users; that is why we decided to introduce visual programming environment *Blockly* for them to be able to write HotDrink code more efficiently.

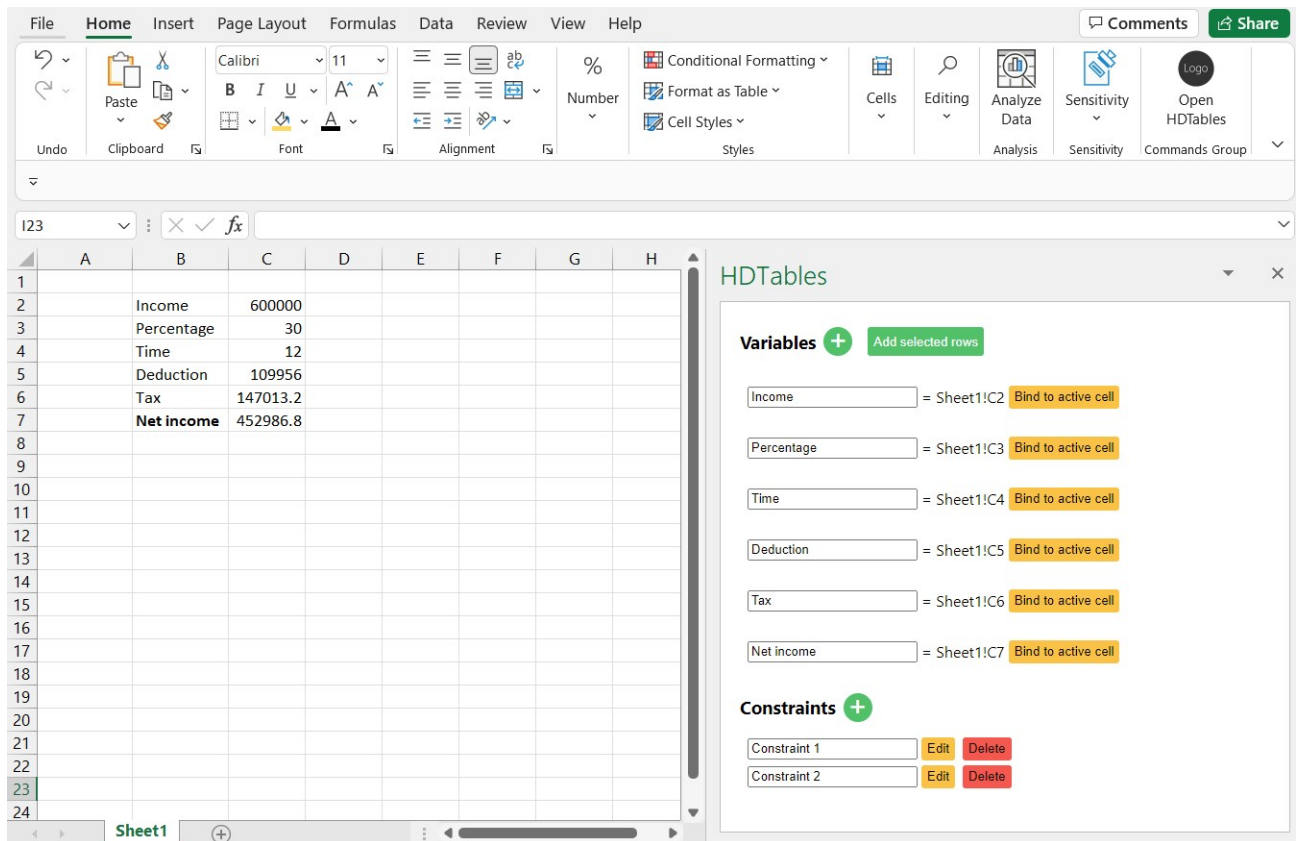


Figure 3.1: Our add-in running in Excel.

When a user clicks *Open HDTables* in the top right corner of Excel (see Figure 3.1), our add-in presents itself on the right side of the spreadsheet. When opening the add-in, the first thing a user sees is the headers *Variables* and *Constraints* and three buttons. Users can add new variables by clicking the plus sign next to the Variables header. By doing so, a new variable named *a* will appear under *Variables*. Doing it multiple times creates variables with the following letter in the alphabet. A user can also add new variables by selecting a single column of cells in Excel and clicking *Add selected rows*. Clicking *Add selected rows* adds the selected number of cells as variables in the add-in, with the same name as the selected cells.

Variables in the add-in come inside input fields where users can change the names to ones that make more sense and are easier to read. Next to the variables, there is a button (*Bind to active cell*), which binds the variable to the currently active cell, and an equality sign with the bound cell will appear next to it.

Clicking the plus sign next to the *Constraint* header adds a new constraint under the header. Each constraint starts with the name *Constraint* with a number in ascending

order, beginning at 1, with the possibility to change it to something more explaining. Next to each constraint is an *Edit* button and a *Delete* button. The *Edit* button presents the user with the Blockly workspace, as can be seen in Figure 3.2, and the *Delete* button deletes the constraint, both from the view and the view-model.

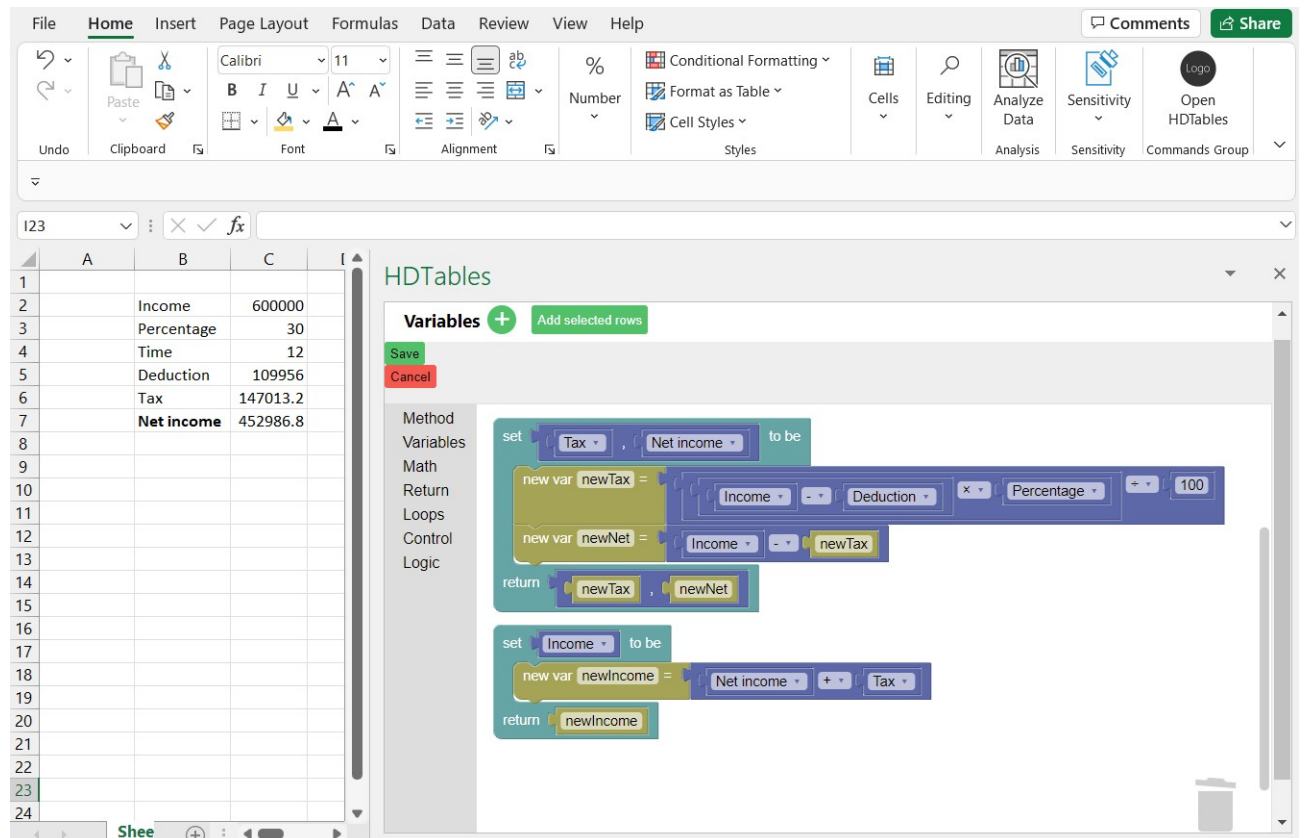


Figure 3.2: Blockly code for the example in Figure 3.1.

The Blockly workspace comes with a toolbox where all the different blocks are categorized on the left and an empty canvas where users can assemble blocks on the right. The different categories inside the toolbox are *Method*, *Loops*, *Control*, *Logic*, *Variables*, *Math*, and *Return*. Hovering over any block will display a tooltip to describe how the block works.

There is only one block inside the *Method* category (shown in Figure 3.3). This block (called *method block*) should be the outermost block in the constraint. However, multiple method blocks can be placed next to each other in the same constraint. The method block has three placeholders where users can place blocks. The top placeholder is where the variables that will be updated are placed. The center placeholder is where users assemble the code to update the variables. Lastly, the bottom placeholder is where users

return the newly computed values of the variables. The top and bottom placeholders can contain multiple variables. However, they should always contain the same amount of variables, and the returned values should match the position of the variable it updates.

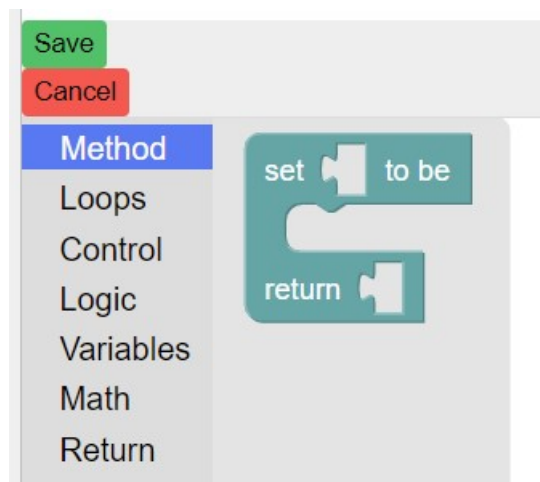


Figure 3.3: Example of the method block inside the Method category.

Errors in the Blockly code will result in different error messages at the top of the Blockly workspace depending on the error. Having a block that is not a *method* block as the outer block will display one error message. Another error message is displayed by leaving the top placeholder of a *method* block empty, and another by leaving the bottom placeholder empty. Lastly, having a different amount of variables in the top and bottom placeholders will display another error message.

Figure 3.4 shows the contents of the Variables category. The top block in this category (called a *variable block*) is the most important one; it lets users choose one of the variables they have defined from a dropdown menu (see Figure 3.5). The variable block can be placed in the top placeholder of a method block and will then be the variable to be updated by the method, or it can be used inside the center placeholder to access its value. The second block has an input field for a number and returns that number. The third one is a block with two placeholders, each accepting only variable blocks or more of itself. This block is used in the top or bottom placeholders of the method block when the user wants to update multiple variables in a single method. Lastly, are three blocks that are related to each other. The first lets the user make a new temporary variable and assign a name and value to it. The second lets the user change the value of an already defined temporary variable. The third returns the value of a temporary value.

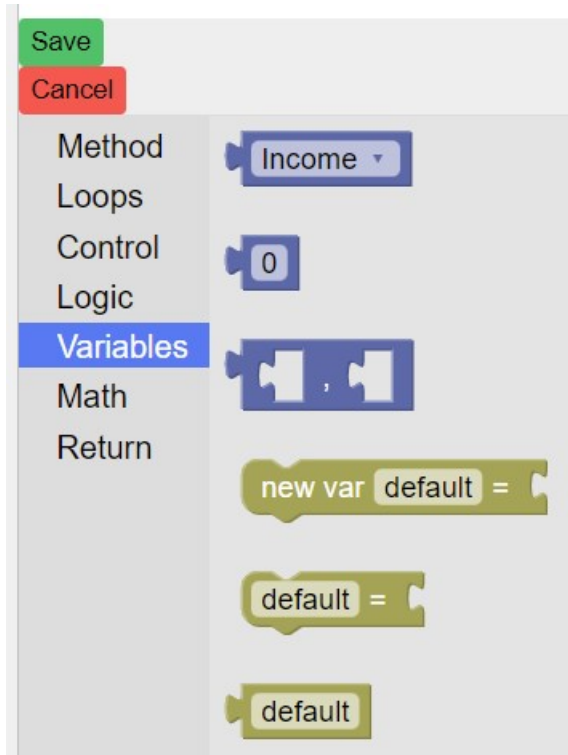


Figure 3.4: Example of the contents in the Variables category.

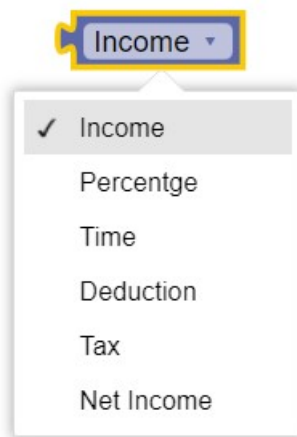


Figure 3.5: Example of a variable block.

After the user has created all the desired methods in a constraint, they can either click *Cancel* or *Save*. Clicking *Cancel* exits the constraint editor and discards all the changes made, and clicking *save* also exits the constraint editor but saves the changes. The constraint will then be in place, such that changing values of cells bound to a variable in the constraint will trigger the constraint to update variables accordingly. The user can

then go back to edit the constraint if needed, resulting in an updated constraint. Clicking the *delete* button next to a constraint removes that constraint from the add-in window while also removing it from the constraint system. The changes will not propagate to bound variables if the user changes the value of a variable in a deleted constraint.

# Chapter 4

## Implementation

While HotDrink has initially been targeted at experienced programmers making web applications with predefined constraints for end-users to utilize, our implementation takes this a step further, letting end-users define their own constraints using our add-in in Excel. We developed our implementation as an Excel add-in using the Office JavaScript API. The entire project can be found in the HDTables [6] GitHub repository.

### 4.1 Architecture

We want to add constraints to spreadsheets. To achieve this, we create a web app in the form of an Office add-in in Excel and integrate it with the HotDrink API. HotDrink needs a way to interact with cells in Excel for the add-in to work. The Office.js JavaScript API lets the add-in interact with objects in Excel, thus letting us read from and write to cells in a spreadsheet.

Creating a variable using the add-in connects that variable to a HotDrink variable. Changing the value of a cell bound to a variable in the add-in will trigger an update on the corresponding HotDrink variable. Conversely, changing the value of a HotDrink variable will trigger an update in the cell connected to the variable.

We use the visual programming environment Blockly for users to be able to make constraints with HotDrink. The Blockly workspace has access to all variables defined by the user in the add-in. Adding new variables will add them to Blockly, and renaming them will rename them in Blockly. Figure 4.1 depicts the architecture of our project.



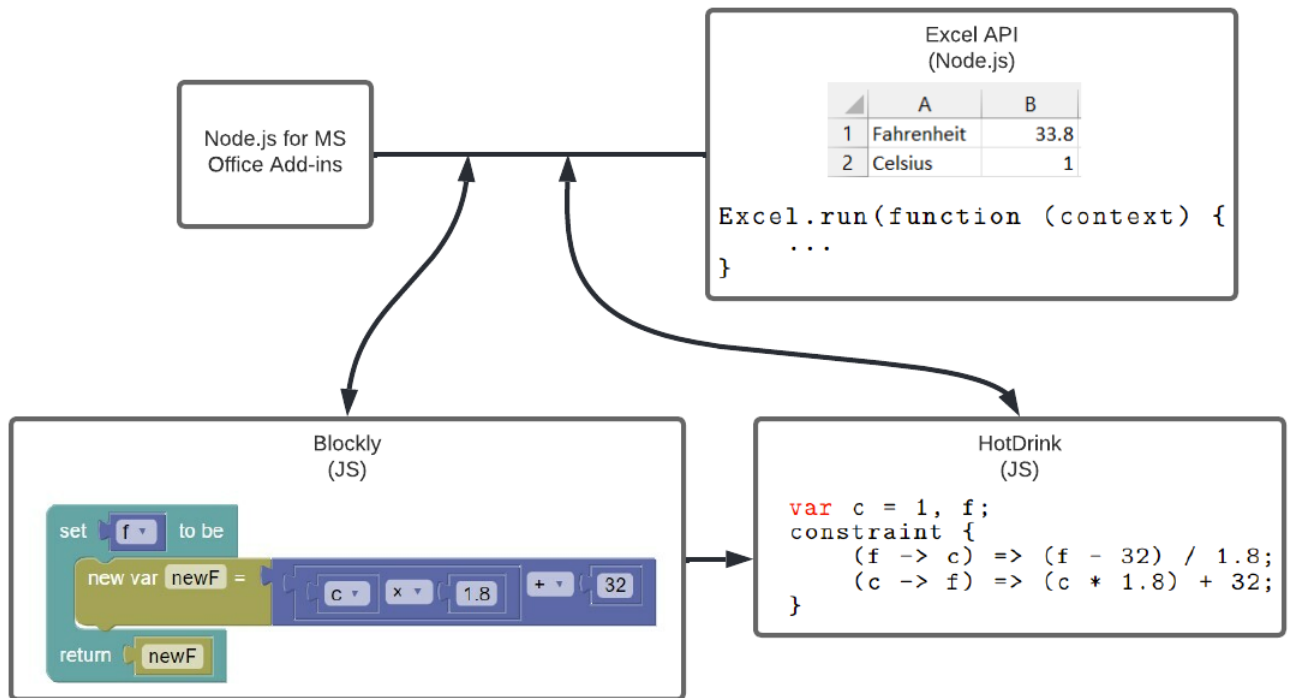


Figure 4.1: Architecture of our prototype implementation.

## 4.2 Add-in implementation

Office Add-ins can be created using either the Yeoman generator for Office Add-ins or Visual Studio [2]. We decided to use the Yeoman generator to develop our project because it generates a Node.js project that can be managed with any editor. We used Visual Studio Code as our editor.

The Office.js JavaScript API gives us access to the `workbook` object in Excel, which lets the add-in interact with Excel. The workbook stores collections of tables, worksheets, and more. Excel data can be accessed and changed through this. Furthermore, the workbook lets the add-in add, navigate through and assign event handlers to worksheets. Event handlers are necessary in order for our add-in to work with HotDrink.

Figure 4.1 shows the code for adding an event handler to the selected cell. The function `addFromSelectionAsync()` takes a string `id` as an argument which is the id used to refer to the binding later on. If adding the event handler succeeds, we retrieve a reference to the selected cell by using the `getActiveCell()` method. Each cell contains much information,

which would require some time to load. To avoid loading all information from a cell, we need to use the `load()` function on the cell to get the information we need. When binding a variable to a cell, we want to know which cell has been bound to the variable. Calling `load("address")` on the cell loads the cells address. It is an asynchronous function, meaning that we have to call `context.sync()` to ensure the information is loaded. After `context.sync()` has finished, we retrieve the address, which would be something like `Sheet1!A1` and add it next to the variable bound to the cell.

Listing 4.1: Code for adding an event handler on the selected cell.

```

1  Office.context.document.bindings.addFromSelectionAsync(
2      ↪ Office.BindingType.Text, { id: id }, function (asyncResult) {
3      if (asyncResult.status == Office.AsyncResultStatus.Failed) {
4          ...
5      } else {
6          Excel.run(function (context) {
7              var activeCell = context.workbook.getActiveCell();
8              activeCell.load("address");
9
10             return context.sync().then(function () {
11                 document.getElementById(`${id}cell`).innerHTML = ` =
12                 ↪ ${activeCell.address}`;
13             });
14         }).catch((e) => {
15             ...
16         });
17     });

```

### 4.3 Blockly implementation

Listing 4.2 shows an example of creating a custom block using Blockly. The example shows how the bottom block from Figure 3.4 can be created. When creating blocks, we can define all types of connections the block contains, the text on the block, the color, the tooltip, and more. Lines 4 to 8 in the example show how we have designed the block — lines 12 to 14 show how we assemble and return the code. In the example, we only want the text from the input field, and we get it by calling `block.getFieldValue("TEMP_VAR")`. We return a tuple with the value we want to return and a number. The number decides the operator precedence of the value, 0 being the lowest.

Listing 4.2: Code to setup and assemble the code from a block.

```

1 export function setupTempVarGetter() {
2   Blockly.Blocks["temp_var_getter"] = {
3     init: function () {
4       this.appendDummyInput().appendField(new
5         ↪ Blockly.FieldTextInput("default"), "TEMP_VAR");
6       this.setOutput(true, null);
7       this.setColour(60);
8       this.setTooltip("Gives the value of the named temporary
9         ↪ variable");
10      this.setHelpUrl("");
11    },
12  };
13  JavaScript["temp_var_getter"] = function (block) {
14    let name = block.getFieldValue("TEMP_VAR");
15    return [name, 0];
16  };
17 }

```

The Blockly workspace contains a toolbox where users can get all the different blocks. The toolbox can be organized with categories, as seen on the left side of Figure 3.1. Listing 4.3 shows how some of the *Variables* category in Figure 3.1 is specified. We create a category for each type of block. Calling `setupTempVarGetter()` from Listing 4.2 creates the block and adds it to the toolbox. Such *setup* methods are called for each custom block we create. Line 2 in Listing 4.2 sets the *type* of the block, placing the block where it matches the type in the toolbox.

Listing 4.3: Code to setup a toolbox in Blockly

```

1 export function setupToolbox() {
2   var toolbox = {
3     kind: "categoryToolbox",
4     contents: [
5       {
6         kind: "category",
7         name: "Variables",
8         contents: [
9           ...
10          {
11            kind: "block",
12            type: "temp_var_update"
13          },
14          {
15            kind: "block",
16            type: "temp_var_getter"
17          },
18        ],
19      },
20      ...
21    ]
22  }
23  Blockly.inject("blocklyDiv", { toolbox: toolbox });
24 }

```

## 4.4 Code generation

Method blocks contain three fields: top, center, and bottom; we refer to these as the output, code, and return fields, respectively. The output field is where output variables reside. Output variables are all the HotDrink variables we want the method to affect. The code field is where users write the code to update the output variables. The return field is where the new variables made in the code field are returned to update the output variables.

When assembling the code from method blocks, we start by getting all the input variables. The input variables are all the HotDrink variables in a method that affects the output variables. To get all the input variables, we go through all blocks inside the code and return fields and filter out all unique HotDrink variables. We can do this by calling `getDescendants()` on the code field in the method block, which returns a list of all blocks nested in the field. We can then match the nested blocks to variable blocks and filter out all unique variables. Then, we collect all output variables from the output field by the same method. Then, we extract the code, collect the return values and append them next to a return statement to the code. When all fields are collected, we assemble and return them in a JSON format as seen in Listing 4.4.

Listing 4.4: JSON returned from method block

```
1 return '{
2     "inputs": [${uniqueInputs}],
3     "outputs": [${uniqueOutputs}],
4     "code": "${code}"
5 },'
```

To create a constraint, we start by retrieving all code from the most recently saved Blockly workspace. If the user has assembled the Blockly code correctly, the generated code will be as in Figure 4.4, with one for each method in the constraint. We add the code to a list in a JSON-like object and parse it to JSON. We parse the code inside a *try/catch* block, and thus if the parsing fails, the user has not correctly assembled the code, resulting in an error message on the top of the Blockly workspace. If the parsing succeeds, we go through all input and output variables and create a list of all unique variables.

Figure 4.5 shows how HotDrink methods are created. The *Method* object takes as arguments the number of unique variables in the constraint, a list of all input positions, a list of all output positions, masktype, and a function with the code of the method.

We loop through methods in the constraint, and for each method, we create a list with positions of all input variables and a list with all output variables in the list of unique variables. When adding the function argument, we create an arrow function as a string that we evaluate with the `eval()` function. We create the arrow function with the inputs joined by `,` on the left side of the arrow and the code on the right. If the constraint already exists, we remove the existing component from the system and create a new component. We add all constraints from the old component to the new with an updated version of the edited constraint. If the constraint does not exist, we create a new constraint and add it to the component.

Listing 4.5: JSON returned from method block

```

1 const methods = code["methods"].map((method) => {
2   const inPositions = method.inputs.map((inn) =>
3     ↪ allVars.indexOf(inn));
4   const outPositions = method.outputs.map((out) =>
5     ↪ allVars.indexOf(out));
6   return new Method(allVars.length, inPositions, outPositions,
7     ↪ [maskNone], eval(`(${method.inputs.join(",")} => {
8     ↪   ${method.code}
9     ↪ }`))
10  });

```

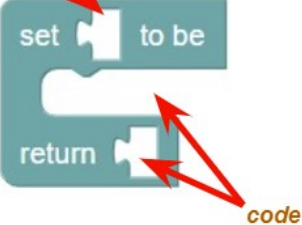
Table 4.1 shows the correspondence between our tool, HotDrink code, and the JavaScript/HotDrink API. The first row shows a method block in the first column. We extract output variables, code, and input variables from the method block. The second column shows the information extracted from a method block in HotDrink code. The last column shows how we construct a method with the HotDrink API in JavaScript.

The second row in the table shows two variables and two constraints in our prototype, what a component looks like in HotDrink code, and how to create a constraint system and add a component to it using the HotDrink API in JavaScript.

The third row in the table shows two variables in our prototype, what variables look like inside a component in HotDrink code, and how to add the variable to the component using the HotDrink API in JavaScript.

The last row in the table shows two constraints in our prototype, what constraints look like inside a component in HotDrink code, and how to create a constraint and add it to the constraint system using the HotDrink API in Javascript.

Table 4.1: Correspondence between our tool, HotDrink code, and JavaScript/HotDrink API.

<p><i>output variables</i></p>  <p><i>input variables</i> are extracted from <i>code</i></p>	<pre>constraint {   (<i>input variables</i> -&gt;    <i>output variables</i>) =&gt; <i>code</i>; }</pre>	<p><b>new</b> Method(<i>amount of input and output variables</i>, <i>positions of input variables</i>, <i>positions of output variables</i>, [maskNone], (<i>input variables</i>) =&gt; {<i>code</i>});</p>
<p><b>Variables</b> + Add selected rows</p> <p>a Bind to active cell</p> <p>b Bind to active cell</p> <p><b>Constraints</b> +</p> <p>Constraint 1 Edit Delete</p> <p>Constraint 2 Edit Delete</p>	<pre>component comp {   <i>variables</i>   <i>constraints</i> }</pre>	<p><b>const</b> system = defaultConstraintSystem;  <b>let</b> comp = new Component("Component");    system.addComponent(comp);</p>
<p><b>Variables</b> + Add selected rows</p> <p>a Bind to active cell</p> <p>b Bind to active cell</p>	<pre>component comp {   <i>variables</i>   <i>constraints</i> }</pre>	<p>comp.emplaceVariable(<i>name of variable</i>, <i>initial value</i>);</p>
<p><b>Constraints</b> +</p> <p>Constraint 1 Edit Delete</p> <p>Constraint 2 Edit Delete</p>	<pre>component comp {   <i>variables</i>   <i>constraints</i> }</pre>	<p><b>const</b> cspec = new ConstraintSpec(Array.from(methods));</p> <p>comp.emplaceConstraint(<i>id of the constraint</i>, cspec, <i>variables in the constraint</i>, false);</p>

# Chapter 5

## User study

In a think-aloud study, we asked ten participants to create a constraint in Microsoft Excel using our add-in. The constraint we asked them to create is a constraint with Fahrenheit and Celsius, like in Listing 2.1. We separated the participants into two groups: one group of five experienced programmers and one group of five non-programmers.

We gave the two groups different amounts of information to complete the task. The experienced programmers were tasked to create a constraint with Fahrenheit and Celsius without any additional information. The non-programmers got the same task with the additional information that they only needed to create two variables, one constraint, and a method block should be the outermost block in a constraint. The steps we wanted the participants to complete with this information are:

1. Create two variables.
  - 1.1. Bind the variables to different cells.
  - 1.2. Change the name of the variables to more explainable ones (*optional*).
2. Add only one constraint
  - 2.1. Change the name of the constraint to a more explainable one (*optional*).
  - 2.2. Click *Edit* on the constraint.
  - 2.3. Use Blockly to create two correct methods.
  - 2.4. Save the constraint.
3. Change the value of both cells to see if the constraint works.

## 5.1 Experienced Programmers: Results

We created a table with each participant as the rows and each step we wanted them to complete as the columns, as shown in Figure 5.1. The cells can contain one of three colors: green, red, or yellow. Green represents that the participant could complete the step without any help (whole dots), yellow represents that we gave some input to help them understand how to complete the step (half dots), and red represents that they were not able to complete the step (empty dots).

	1	1.1	1.2	2	2.1	2.2	2.3	2.4	3
Participant 1	●	◐	●	◐	●	●	●	●	●
Participant 2	●	●	○	●	○	●	●	●	●
Participant 3	●	●	●	◐	○	●	●	●	●
Participant 4	●	●	●	◐	●	●	●	●	●
Participant 5	●	●	●	◐	●	●	●	●	●

Figure 5.1: Performance on each task for experienced programmers

The figure above shows that every participant successfully created the constraint. Some participants could not complete either steps 1.2 or 2.1 or both. However, they could still complete the entire task, as steps 1.2 and 2.1 are optional. Several participants started by creating one constraint with one method and then wanted to create another constraint with the other method. In those instances, rather than letting them waste time creating another constraint, we explained that they could complete the entire task using only one constraint, resulting in successfully creating the other method in the same constraint.

To become acquainted with the tool, some participants attempted to create a variety of variables and constraints. Some participants believed they could place blocks in the top right corner of a method block and were surprised when this did not work. However, after some trial and error, they figured out where to place the blocks.



All of the participants demonstrated a clear understanding of how to add variables and constraints. They expressed that Blockly was intuitive and simple to use, and they had a positive overall experience with the add-in.

## 5.2 Non-programmers: Results

As mentioned before, non-programmers performed the same tasks as programmers. We created a similar table to the one in section 5.1 to map the results of the non-programmers, as shown in Figure 5.2 below.

	1	1.1	1.2	2	2.1	2.2	2.3	2.4	3
Participant 1	●	●	○	☉	○	●	●	●	●
Participant 2	●	●	○	●	○	●	☉	●	●
Participant 3	●	●	●	●	○	●	☉	●	●
Participant 4	●	●	●	●	○	●	☉	●	●
Participant 5	●	●	○	●	○	●	☉	●	●

Figure 5.2: Performance on each task for non-programmers

From the figure above, we can see that all the non-programmers were able to create only two variables (as expected from their instructions) and bind them to the correct cells. Only one participant tried to create multiple constraints, despite being instructed only to make one. None of the participants came up with new and illuminating explanatory names for the constraint. All but one participant had difficulty understanding how blocks worked and required some assistance. We assisted them by asking them questions about what they wanted to accomplish and gently guiding them in that direction.

Like some of the experienced programmers, one participant added several constraints and variables to familiarize themselves with the add-in.

All of the non-programmers were, in the end, able to create the constraint and expressed that the add-in was relatively easy to understand after getting familiarized with it.

## 5.3 Feedback

The participants who created several variables and constraints to familiarize themselves with the tool expressed a desire to be able to delete variables in the same way that constraints could be deleted. We discuss why variables cannot be deleted in the future work section.

Several of the non-programmers said that it would be helpful to have some sort of tutorial going through every step of creating a constraint before trying the add-in.

## 5.4 Summary

All participants, both programmers and non-programmers were able to complete the task. We see that most of the programmers gave descriptive names to the constraint and variables, while none of the non-programmers gave descriptive names to the constraint, and few gave to variables. We assume this is because experienced programmers are used to the concept of variables and giving them descriptive names, while non-programmers are not.

We can also see that all experienced programmers were able to assemble method blocks without assistance, whereas most non-programmers required some sort of assistance to understand it. We believe this is because programmers understand the fundamentals of programming while non-programmers require some time to become acquainted with it.

In conclusion, the results from this user test show promise, as all participants were able to create the constraint, and the participants gave positive feedback on how they experienced our solution.

# Chapter 6

## Related work

This chapter looks at several studies related to block-based programming and spreadsheets. Some of these studies show how visualization tools improve the coding experience of end-users. Many studies show that spreadsheets are error-prone and generally lack important functionalities that could benefit spreadsheet users. Some studies also investigate tools to address problems with the spreadsheet paradigm. A few tools add to existing spreadsheet programs like Excel, while others introduce a new spreadsheet software with the tool integrated.

### 6.1 XLBlocks

When writing formulas in Excel, it can be easy to misplace parenthesis, quotes, and commas. Research has shown that block-based environments can improve the performance of beginner programmers. Jansen and Hermans [23] hypothesize that a block-based formula editor also can be beneficial for spreadsheet users as it decreases the possibility of syntax errors.

XLBlocks, like our prototype, is an Excel Add-in developed with the Excel JavaScript API that uses Blockly to generate Excel formulas. XLBlocks has extended the Blockly library with custom blocks and a code generator to define and generate spreadsheet formulas. The functions included in their research prototype are: SUM, SUMIFS, INDEX, IFERROR, IF, MATCH, VLOOKUP, -, /, <, and >. These functions were chosen based on the frequent use of functions according to the Enron corpus [19].

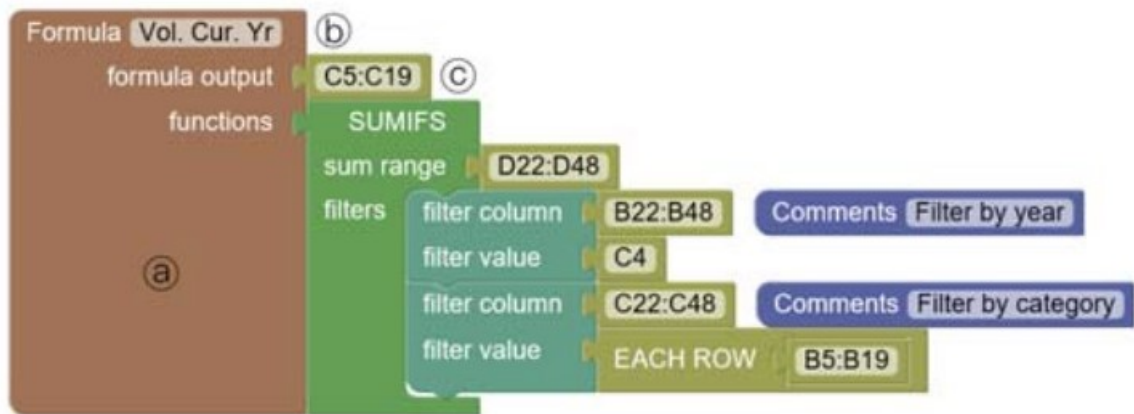


Figure 6.1: Example of a block definition of a SUMIFS formula [23]

When using XLBlocks, a user begins with a formula block ((a) in Figure 6.1). The formula can be given a name by the user (b), an output range (c), and the functions used in the formula. The two comments in the example will not be shown in the spreadsheet but are available in the block editor to describe the code to the programmer as with text-based programming.

Jansen and Hermans aim to determine how a block-based formula editor could support professional spreadsheet users while developing or maintaining formulas. To address this goal, they developed XLBlocks and conducted a think-aloud study where participants completed eight common spreadsheet tasks. When the participants completed the tasks, they interviewed them and asked them to evaluate the XLBlocks interface using the Cognitive Dimensions of Notation (CDN) framework [10]. For each dimension, they asked them to answer the two following research questions:

- RQ1: "What are the benefits of XLBlocks regarding this dimension?"
- RQ2: "What are the drawbacks of XLBlocks regarding this dimension?"

The participants included in this study all use Excel professionally, with an average of 20 years of experience using Excel. The participants ranked themselves on a scale from one to ten on how experienced they were in Excel and scored an average of eight out of ten.

Jansen and Hermans results show that XLBlocks received better evaluation on all dimensions of the CDN framework. The dimensions: *Error-proneness*, *Secondary notation*,

*Provisionality*, *Premature commitment*, and *Progressive evaluation* showed the most notable difference.

The participants realized that considering the correct syntax of functions in XLBlocks was unnecessary. Furthermore, parts of formulas were easier to edit in XLBlocks as parts of the formula can effortlessly be dragged and dropped onto the canvas.

When comparing this project to ours, we see that both projects add functionality to Excel, improving the end user's experience. XLBlocks is, as our prototype, an add-in for Excel, and both add-ins incorporate the visual programming environment Blockly to let end-users code more intuitively.

## 6.2 Block Shelves

Block shelves [21] is a formatting and organizing tool developed to address some of the limitations in block editors such as Blockly. Three of these limitations are program structure, readability, and re-use.

Block shelves allow a user to group and arrange collections of blocks on shelves in a shelfbox to get a better overview of the block code. It comes with seven primary shelf functions:

1. Show/hide. Enables users to show or hide blocks on a shelf. If a user clicks the hide button on a shelf, the corresponding blocks will disappear from the canvas.
2. Collapse/expand. Enables users to collapse all blocks on a shelf into one block.
3. Enable/disable. Enables users to enable/disable block code in a shelf.
4. Duplicate. Enables users to duplicate all the blocks on a shelf.
5. Comment. Lets users leave a comment on a shelf.
6. Delete. Lets a user delete a shelf without deleting the corresponding blocks.
7. Export/import. Makes it possible to share blocks across projects in the form of XML.

Hsu et al. describe [21] how they tested their tool on 60 graduate and undergraduate students from universities in Taiwan, all with some experience with programming but no one who had ever used a visual programming language (VPL). The study aimed to measure how block shelves improve upon search and reading time, block code navigation, and whether it improved block code understanding when added to a block editor compared to a block editor without the tool.

The results from the paper indicate that block shelves significantly improve the readability of projects and the time it takes to write one. Moreover, code block understanding and navigation of projects became much easier once the users had become accustomed to the tool for large projects.

Block Shelves, like our prototype, is a visual environment made to improve users' experience. Our prototype improves end-users' experience of Excel by letting them code using the visual programming environment Blockly, and Block Shelves goes one step up by adding to block environments, improving the block coding experience.

### 6.3 Polaris

Polaris is an Excel Add-in developed using VBA (Visual Basic for Applications) for context-aware navigation in spreadsheets [22]. Polaris monitors the cells users select and uses that information to "guess" where it should navigate when a user switches between sheets.

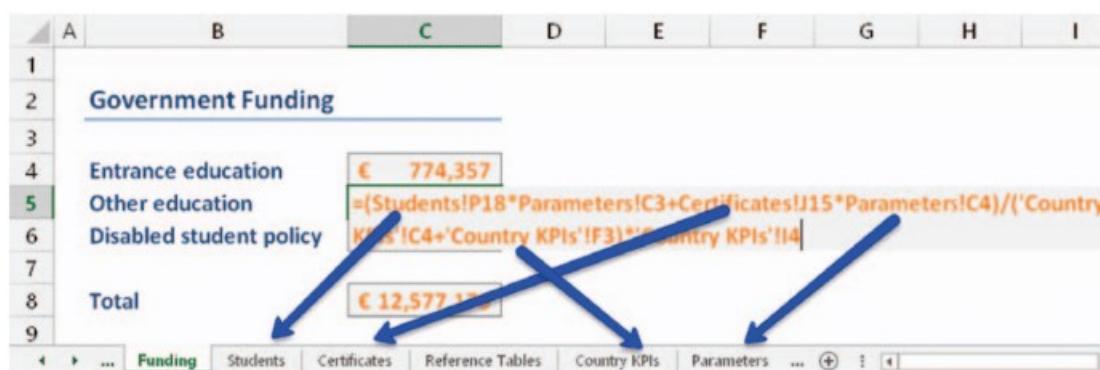


Figure 6.2: Example of a formula that makes reference to multiple worksheets [22].

In Figure 6.2, a user has selected cell C5. If the user then, for example, switches to sheet Certificates, Polaris will go through the references of the last selected cell and

see if it finds any of them in Certificates. In this example, it will find the reference Certificates!J15 and activate cell J15.

Polaris and our prototype are both add-ins for Microsoft Excel created to improve the experience of spreadsheet users.

## 6.4 TrueGrid

Spreadsheet editors like Excel lack basic editor services such as reference resolution and syntax highlighting, while modern IDEs do not. However, most IDEs lack the interactive and live interaction style that spreadsheet users want. TrueGrid [20] introduces the concept of bridging the gap between spreadsheets and programming. Hermans and van der Storm use JavaScript in their prototype to illustrate that one can program a spreadsheet-like grid using TrueGrid. The key feature of TrueGrid is that programmers can see the code and data simultaneously; it also is live like a spreadsheet, meaning that changing code or data updates the grid.

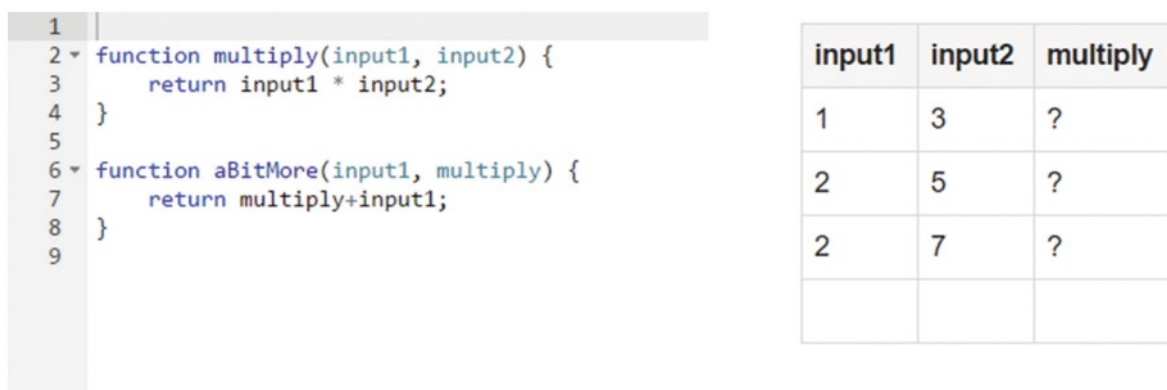


Figure 6.3: Example of a TrueGrid implementation, with the code editor on the left and the grid on the right side [20].

While TrueGrid introduces a new spreadsheet and our prototype is an add-in to the existing spreadsheet software Excel, both projects introduce a new way of coding in spreadsheets, adding previously non-existing functionalities.

## 6.5 ZenSheet Studio

ZenSheet Studio [17] is a web client that provides a generalized spreadsheet computing experience. It is a part of the ZenSheet project, which aims to turn spreadsheets into modern and robust computing environments with minimal influence on the learning curve. ZenSheet can be seen as a real-time data visualization tool, a live IDE for a lazy programming language, or a spreadsheet environment that intuitively delivers the generalized modeling power of ZenSheet. The project's approach is motivated by the idea of adding a spreadsheet-like interface to a functional programming language like Haskell [27].

Initial tests of ZenSheet Studio with students and industry practitioners have shown promise. However, extensive testing must be conducted in the future for conclusive results.

ZenSheet Studio presents a spreadsheet environment with new functionalities, while our prototype adds new functionalities to an existing spreadsheet environment. However, both projects aim to further the experience of using spreadsheets.

## 6.6 Spreadsheet Functional Programming

Wakeling [42] introduces a way to define Haskell functions in Excel. When a user adds a comment to a cell and begins the comment with *Haskell*, the program will interpret the following lines as Haskell code. When a user changes the worksheet, the implementation will execute some VBA code that writes a calculations file, executes a Haskell interpreter, and reads a results file. The calculations file contains a Haskell program that the Haskell interpreter executes, and the results file contains a list with pairs of results and cell references. After reading the results file, the program updates the cells with correct values.

Both Wakeling and us create an add-in to Microsoft Excel that introduces another form of programming and adds functionalities previously unavailable to spreadsheets.



## 6.7 Gradual Structuring

Learning to program in spreadsheets is easier than in traditional programming languages. However, spreadsheets are not as expressive as traditional programming languages, limiting the levels of computational thinking that can be taught using spreadsheets. Gradual Structuring [31] is a concept proposed by Miller et al., and it is the idea of introducing language features to spreadsheets, enabling more structured models in spreadsheets.

The ability to group cells locally and apply a single formula to them is necessary to enable Gradual Structuring. This feature, termed *cell grouping*, can eliminate the error-prone use of copy-pasting in spreadsheets and makes the structure of spreadsheets more explicit. Cell grouping introduces a range of new features. Miller et al.'s paper focus on the learnability and usability of these features.

Gradual structuring and our prototype both introduce new functionalities to spreadsheets by adding a form of programming to them.

## 6.8 Object Oriented Functional Spreadsheets

While spreadsheets have become one of the most successful and used computer applications, they are also limiting and frustrating. Clack and Braine [11] believe that the computational model of spreadsheets can be both extended to deliver more functionality and simplified to benefit non-programmers. They propose a new spreadsheet paradigm that includes object-oriented programming features such as overloading, inheritance, subsumption, a class hierarchy, and dynamic despatch on an object. Their system also includes many functional programming features such as a strong type system, higher-order functions, curried partial applications, lazy evaluation, and referential transparency.

As mentioned, Robert and Roger propose a new spreadsheet paradigm to increase the functionalities of spreadsheets; this is similar to our project in that we also introduce new functionality to spreadsheets.

## 6.9 SpreadsheetDoc

Spreadsheet systems lack a proper setup to document spreadsheet programs. It is possible to add general notes to cells in spreadsheets, but not in a structured way as with tools like JavaDoc. A user taking over an existing Excel program and trying to understand everything can be very cumbersome, often resulting in them asking colleagues for help or even quitting the task. SpreadsheetDoc [12] is an Excel Add-in that allows end-users to add documentation to their projects more structurally. The core functionalities of the add-in are the ability to document an entire spreadsheet document, each worksheet, single cells, columns, rows, ranges, input cells, and output cells.

SpreadsheetDoc and our prototype are both Microsoft Excel add-ins that add functionality to spreadsheets that were previously unavailable.

## 6.10 ClassSheets

The cell-oriented, low-level development process of spreadsheets can have many errors. ClassSheet [16] is a higher-level object-oriented model that improves upon this process. By integrating concepts from the Unified Modeling Language (UML), ClassSheets allow users to express business object structures inside a spreadsheet explicitly. The approach of ClassSheet is to link the object-oriented modeling world with spreadsheet applications.

Our approach is similar to that of ClassSheet in that both projects introduce a higher-level development process for spreadsheets in order to address some missing functionalities in spreadsheets.

## 6.11 Tabula

Tabula [29] is a modeling language for spreadsheets inspired by the visual notation of the ClassSheet modeling language [16]. Tabula, on the other hand, presents more expressive attributes than ClassSheet, such as type constraints and nested classes with repetitions enabled by a different abstract representation. Furthermore, Tabula includes a bidirectional transformation engine that ensures synchronization when updating the spreadsheet or the Tabula model.

Tabula is similar to our project in the same way that ClassSheet is similar. The bidirectional transformation engine that ensures synchronization is similar to how changing the values of variables in a constraint synchronizes the corresponding variables.

# Chapter 7

## Discussion and Future Work

Microsoft Excel is a prevalent spreadsheet software widely used across industries; it is oftentimes used to fill out various forms. In order to validate the correctness of the entered data values and to improve users' experience with such forms within spreadsheets, Excel-integrated Visual Basic for Applications and other approaches are frequently layered on top of Excel spreadsheets. However, even with these extra layers of validations, spreadsheets are still very prone to errors.

Our approach is to introduce some *structure* to Excel. We accomplish this by introducing HDTables, a tool that allows users to define the *multi-way* data-flow between cells in Excel explicitly. This is in contrast to the traditional Excel model, where specifying multi-way data-flow is not possible. We have used the multi-way data-flow constraint system library HotDrink to enable such explicit definitions. The library requires some knowledge of JavaScript, which may be impeding for some of the Excel users. We tackle this problem by letting users define constraints using the visual programming environment Blockly. Block-based environments such as Blockly let users write code using blocks. Blocks are beneficial because they come in various shapes and colors, making it intuitive for non-programmers to understand how pieces connect. This is expressed by the participants of our user study. Our prototype implementation shows the feasibility of specifying multi-way data-flow constraint systems directly in spreadsheets targeted at non-developer users.

We have identified several directions for future work. First, constraint system libraries other than HotDrink could be adapted and integrated with Excel—or, for that matter—other spreadsheet software. These constraint systems may be based on languages other

than JavaScript, hence providing support for them is needed, either by creating a Blockly-based (or another block-based) “front-end” for them, or by employing an entirely different visual programming environment (such as, for example, node-based environments). Trying out other visual programming specifications could give some insight into what end-users need to be best able to create constraints.

In order to adequately support an ever-growing community of spreadsheet users, implementing functionality to transform (legacy) spreadsheet specifications into HDTable-based spreadsheets seems necessary. This will significantly extend the pool of HDTable’s users and allow conducting a large-scale evaluation of the tool which would be necessary to gain a better understanding of how users interact with “structural spreadsheets”. Ultimately, this will help understand how spreadsheet software can be further improved.



# Bibliography

- [1] Blockly.  
**URL:** <https://developers.google.com/blockly/guides/overview>.
- [2] Build an Excel task pane add-in.  
**URL:** <https://docs.microsoft.com/en-us/office/dev/add-ins/quickstarts/excel-quickstart-jquery?tabs=yeomangenerator>.
- [3] Microsoft Excel, .  
**URL:** <https://www.microsoft.com/en-us/microsoft-365/excel>.
- [4] Excel add-ins overview, .  
**URL:** <https://docs.microsoft.com/en-us/office/dev/add-ins/excel/excel-add-ins-overview>.
- [5] Gnumeric.  
**URL:** <http://www.gnumeric.org/>.
- [6] HDTables.  
**URL:** <https://github.com/TorjusFS/HDTables>.
- [7] Apache OpenOffice Calc.  
**URL:** <http://www.openoffice.org/product/calc.html>.
- [8] About Scratch.  
**URL:** <https://scratch.mit.edu/about>.
- [9] Difference Between CSV and XLS.  
**URL:** <https://toggl.com/track/difference-between-csv-xls/>.
- [10] A. F. Blackwell, C. Britton, A. Cox, T. R. G. Green, C. Gurr, G. Kadoda, M. S. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, C. Roast, C. Roe, A. Wong, and R. M. Young. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In Meurig Beynon, Chrystopher L. Nehaniv, and Kerstin Dautenhahn, editors,

- Cognitive Technology: Instruments of Mind*, volume 2117, pages 325–341. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 9783540424062 9783540446170. doi: 10.1007/3-540-44617-6\_31.  
**URL:** [http://link.springer.com/10.1007/3-540-44617-6\\_31](http://link.springer.com/10.1007/3-540-44617-6_31).
- [11] Christopher Clack and Lee Braine. Object-oriented functional spreadsheets. 01 1998.
- [12] Jácome Cunha and Diogo Canteiro. Spreadsheetdoc: An excel add-in for documenting spreadsheets. In *6th National Symposium on Informatics (INForum 2015)*, 2015.
- [13] Nancy Cunniff and Robert P. Taylor. *Graphical vs. Textual Representation: An Empirical Study of Novices’ Program Comprehension*, page 114–131. Ablex Publishing Corp., USA, 1987. ISBN 0893914614.
- [14] Bob Frankston Dan Bricklin. VisiCalc: Information from its creators, Dan Bricklin and Bob Frankston.  
**URL:** <http://www.danbricklin.com/visicalc.htm>.
- [15] Ruth S. Day. Alternative Representations. In *Psychology of Learning and Motivation*, volume 22, pages 261–305. Elsevier, 1988. ISBN 9780125433228. doi: 10.1016/S0079-7421(08)60043-2.  
**URL:** <https://linkinghub.elsevier.com/retrieve/pii/S0079742108600432>.
- [16] Gregor Engels and Martin Erwig. Classsheets: automatic generation of spreadsheet applications from object-oriented specifications. pages 124–133, 01 2005.
- [17] Monica Figuera. ZenSheet studio: a spreadsheet-inspired environment for reactive computing. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 33–35, Vancouver BC Canada, October 2017. ACM. ISBN 9781450355148. doi: 10.1145/3135932.3135949.  
**URL:** <https://dl.acm.org/doi/10.1145/3135932.3135949>.
- [18] John Freeman, Jaakko Järvi, and Gabriel Foust. HotDrink: a library for web user interfaces. *ACM SIGPLAN Notices*, 48(3):80–83, April 2013. ISSN 0362-1340, 1558-1160. doi: 10.1145/2480361.2371413.  
**URL:** <https://dl.acm.org/doi/10.1145/2480361.2371413>.
- [19] Felienne Hermans and Emerson Murphy-Hill. Enron’s Spreadsheets and Related Emails: A Dataset and Analysis. In *2015 IEEE/ACM 37th IEEE International*



- Conference on Software Engineering*, pages 7–16, Florence, Italy, May 2015. IEEE. ISBN 9781479919345. doi: 10.1109/ICSE.2015.129.  
**URL:** <http://ieeexplore.ieee.org/document/7202944/>.
- [20] Felienne Hermans and Tijds van der Storm. TrueGrid: Code the Table, Tabulate the Data. In Paolo Milazzo, Dániel Varró, and Manuel Wimmer, editors, *Software Technologies: Applications and Foundations*, volume 9946, pages 388–393. Springer International Publishing, Cham, 2016. ISBN 9783319502298 9783319502304. doi: 10.1007/978-3-319-50230-4\_29.  
**URL:** [http://link.springer.com/10.1007/978-3-319-50230-4\\_29](http://link.springer.com/10.1007/978-3-319-50230-4_29).
- [21] Sheng-Yi Hsu, Yuan-Fu Lou, Shing-Yun Jung, and Chuen-Tsai Sun. Shelves: A User-Defined Block Management Tool for Visual Programming Languages. In Regina Bernhaupt, Girish Dalvi, Anirudha Joshi, Devanuj K. Balkrishan, Jacki O’Neill, and Marco Winckler, editors, *Human-Computer Interaction – INTERACT 2017*, volume 10515, pages 335–344. Springer International Publishing, Cham, 2017. ISBN 9783319676869 9783319676876. doi: 10.1007/978-3-319-67687-6\_22.  
**URL:** [https://link.springer.com/10.1007/978-3-319-67687-6\\_22](https://link.springer.com/10.1007/978-3-319-67687-6_22).
- [22] Bas Jansen. Polaris: Providing context aware navigation in spreadsheets. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 228–229, Cambridge, United Kingdom, September 2016. IEEE. ISBN 9781509002528. doi: 10.1109/VLHCC.2016.7739690.  
**URL:** <http://ieeexplore.ieee.org/document/7739690/>.
- [23] Bas Jansen and Felienne Hermans. XLBlocks: a Block-based Formula Editor for Spreadsheet Formulas. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 55–63, Memphis, TN, USA, October 2019. IEEE. ISBN 9781728108100. doi: 10.1109/VLHCC.2019.8818748.  
**URL:** <https://ieeexplore.ieee.org/document/8818748/>.
- [24] Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob Smith. Algorithms for user interfaces. In *Proceedings of the eighth international conference on Generative programming and component engineering, GPCE ’09*, pages 147–156, New York, NY, USA, October 2009. Association for Computing Machinery. ISBN 9781605584942. doi: 10.1145/1621607.1621630.  
**URL:** <https://doi.org/10.1145/1621607.1621630>.
- [25] Leo Kelion. Excel: Why using Microsoft’s tool caused Covid-19 results to be lost.  
**URL:** <https://www.bbc.com/news/technology-54423988>.

- [26] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, aug 1988. ISSN 0896-8438.
- [27] Björn Lisper and Johan Malmström. Haxcel: A spreadsheet interface to haskell. 11 2002.
- [28] Carol McGuinness. Problem representation: The effects of spatial arrays. *Memory & Cognition*, 14(3):270–280, May 1986. ISSN 0090-502X, 1532-5946. doi: 10.3758/BF03197703.  
**URL:** <http://link.springer.com/10.3758/BF03197703>.
- [29] Jorge Mendes and João Saraiva. Tabula: A Language to Model Spreadsheet Tables. 2017. doi: 10.48550/ARXIV.1707.02833.  
**URL:** <https://arxiv.org/abs/1707.02833>.
- [30] Mauricio Verano Merino, Jurgen Vinju, and Mark van den Brand. DRAFT-What you always wanted to know but could not find about block-based environments. *arXiv:2110.03073 [cs]*, October 2021.  
**URL:** <http://arxiv.org/abs/2110.03073>. arXiv: 2110.03073.
- [31] Gary Miller, Felienne Hermans, and Robin Braun. Gradual structuring: Evolving the spreadsheet paradigm for expressiveness and learnability. In *2016 15th International Conference on Information Technology Based Higher Education and Training (ITHET)*, pages 1–8, Istanbul, Turkey, September 2016. IEEE. ISBN 9781509007783. doi: 10.1109/ITHET.2016.7760759.  
**URL:** <http://ieeexplore.ieee.org/document/7760759/>.
- [32] Rajeev Pandey and Margaret Burnett. Is it easier to write matrix manipulation programs visually or textually? an empirical study. In *IEEE Symp. Visual Languages*, pages 24–27, 1993.
- [33] Tie Cheng Mary Pat Campbell Patrick O’Beirne, Felienne Hermans. EuSpRIG Horror Stories.  
**URL:** <http://www.eusprig.org/horror-stories.htm>.
- [34] Marian Petre and Thomas R. G. Green. Learning to read graphics: Some evidence that ‘seeing’ an information display is an acquired skill. *J. Vis. Lang. Comput.*, 4: 55–70, 1993.
- [35] John M. Polich and Steven H. Schwartz. The effect of problem size on representation in deductive problem solving. *Memory & Cognition*, 2(4):683–686, July 1974. ISSN

0090-502X, 1532-5946. doi: 10.3758/BF03198139.

**URL:** <http://link.springer.com/10.3758/BF03198139>.

- [36] H. Rudy Ramsey, Michael E. Atwood, and James R. Van Doren. Flowcharts versus program design languages: an experimental comparison. *Communications of the ACM*, 26(6):445–449, June 1983. ISSN 0001-0782, 1557-7317. doi: 10.1145/358141.358149.

**URL:** <https://dl.acm.org/doi/10.1145/358141.358149>.

- [37] Carmen M Reinhart and Kenneth S Rogoff. Growth in a Time of Debt. *American Economic Review*, 100(2):573–578, May 2010. ISSN 0002-8282. doi: 10.1257/aer.100.2.573.

**URL:** <https://pubs.aeaweb.org/doi/10.1257/aer.100.2.573>.

- [38] Ben G. Rittweger and Eoin Langan. Spreadsheet Risk Management in Organisations. 2010. doi: 10.48550/ARXIV.1009.2775.

**URL:** <https://arxiv.org/abs/1009.2775>.

- [39] D.A. Scanlan. Structured flowcharts outperform pseudocode: an experimental comparison. *IEEE Software*, 6(5):28–36, September 1989. ISSN 0740-7459. doi: 10.1109/52.35587.

**URL:** <http://ieeexplore.ieee.org/document/35587/>.

- [40] Peter Sestoft. *What Is a Spreadsheet*, pages 1–24. 2014.

- [41] Gabriele Tomassetti. Are You Abusing Excel? You Need Something Different.

**URL:** <https://tomassetti.me/excel-and-dsls/>.

- [42] David Wakeling. Spreadsheet functional programming. *Journal of Functional Programming*, 17(1):131–143, January 2007. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796806006186.

**URL:** [https://www.cambridge.org/core/product/identifier/S0956796806006186/type/journal\\_article](https://www.cambridge.org/core/product/identifier/S0956796806006186/type/journal_article).



- [43] K.N. Whitley. Visual Programming Languages and the Empirical Evidence For and Against. *Journal of Visual Languages & Computing*, 8(1):109–142, February 1997. ISSN 1045926X. doi: 10.1006/jvlc.1996.0030.


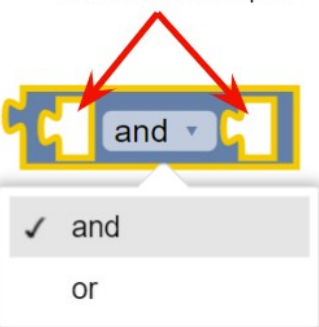
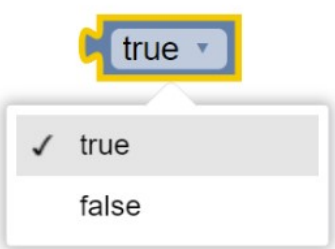
**URL:** <https://linkinghub.elsevier.com/retrieve/pii/S1045926X96900300>.

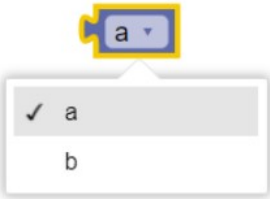
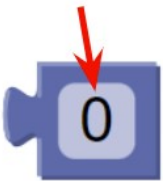
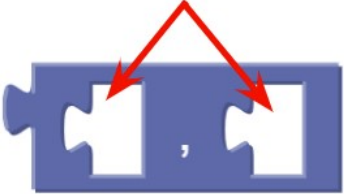

## Appendix A


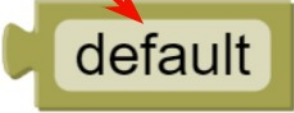
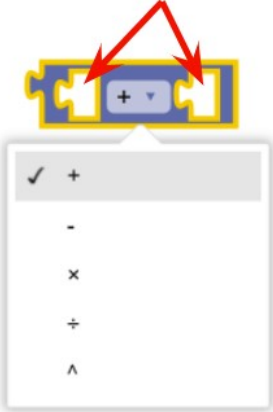

### Code generated from Blockly specifications

Table A.1: Correspondence between visual Blockly specifications in our tool, HotDrink code, and JavaScript/HotDrink API.

<p><i>Number</i> of times to repeat statements in <i>do</i></p>  <p>A green Blockly block with a notch on the left and a bump on the right. The word 'repeat' is on the left, 'times' is on the right, and 'do' is at the bottom. Two red arrows point to the notch and the bump.</p> <p><i>Code</i></p>	<pre>for (var count = 0; count &lt; <i>number</i>; count++) {   <i>code</i> }</pre>
<p>If statement is <i>true</i>, then <i>do</i> some statements</p>  <p>A blue Blockly block with a notch on the left and a bump on the right. It has a gear icon and the word 'if' on the left, and 'do' at the bottom. Two red arrows point to the notch and the bump.</p> <p><i>Code</i></p>	<pre>if (<i>a true or false statement</i>) {   <i>code</i> }</pre>

<p>Values to compare</p>  <p>Returns true if statement is true</p>	<p><i>value1 == value2</i></p> <p><i>value1 != value2</i></p> <p><i>value1 &lt; value2</i></p> <p><i>value1 &lt;= value2</i></p> <p><i>value1 &gt; value2</i></p> <p><i>value1 &gt;= value2</i></p>
<p>Statements to compare</p>  <p><i>and</i> returns <i>true</i> if both values are true  <i>or</i> returns <i>true</i> if one of the statements are true</p>	<p><i>statement1 &amp;&amp; statement2</i></p> <p><i>statement1    statement2</i></p>
 <p><i>true</i> returns <i>true</i>  <i>false</i> returns <i>false</i></p>	<p>true</p> <p>false</p>

 <p>Returns the name of the variable</p>	<p><i>name of the variable</i></p>
<p>Any <i>number</i></p>  <p>Returns the number typed</p>	<p><i>number</i></p>
<p><i>variables</i></p>  <p>Returns a list of two variables</p>	<p><i>[name of variable1, name of variable2]</i></p>
<p><i>Name of the variable</i>      <i>Value of the variable</i></p>  <p>Creates a new temporary variable</p>	<p><b>var</b> <i>name of the variable</i> = <i>Value of the variable</i> ;</p>

<p><i>Name</i> of an existing temporary <i>variable</i></p>  <p>Updated <i>value</i> of the <i>variable</i></p> <p>Updates the value of an existing temporary variable</p>	<p><i>name of the variable</i> = <i>new value</i>;</p>
<p><i>Name</i> of an existing temporary <i>variable</i></p>  <p>Returns the <i>name</i> of an existing temporary var</p>	<p><i>name of a temporary variable</i></p>
<p>Numerical <i>values</i></p>  <p>Returns the mathematical equation of the two <i>values</i></p>	<p><i>value1</i> + <i>value2</i></p> <p><i>value1</i> - <i>value2</i></p> <p><i>value1</i> * <i>value2</i></p> <p><i>value1</i> / <i>value2</i></p> <p><i>Math.pow(value1, value2)</i></p>
<p><i>value</i></p>  <p>Creates a return statement with a <i>value</i></p>	<p>return <i>value</i>;</p>