Triggered Expressions: Event Driven Manipulation of Cell States in Spreadsheets

by

Morgan Voss

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirement for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

© Morgan D. Voss. All Rights Reserved.

The author hereby grants to M.I.T. permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole and in part in any medium now known or hereafter created.

Author: ____________________________

Department of Electrical Engineering and Computer Science

May 26, 2017

Certified by: ____________________________

Daniel J. Jackson

Ford Professor of Artificial Intelligence and Computer Science

Thesis Supervisor

Accepted by: ____________________________

Christopher Terman,

Chairman

Masters of Engineering Thesis Committee
Triggered Expressions: Event Driven Manipulation of Cell States in Spreadsheets

by

Morgan Voss

Submitted to the Department of Electrical Engineering and Computer Science on May 12, 2017, in partial fulfillment of the requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Spreadsheets have become the go-to end-user computing tool for non-technical users, enabling users to create mid to low-level complexity processes for purposes such as administrative tasks, storing data, and reports. In light of this success, spreadsheets developers have worked to expand functionality through a variety of mechanisms, including macros, data analysis and charting, and much more. However, the trend has been to extend functionality through alternative mechanisms that break with the tradition that first made spreadsheets popular. By creating a functional paradigm for event driven manipulation of cell states, and maintaining consistency with the original functional paradigm of the spreadsheet, users can express an expanded set of functionalities without defaulting to alternative programming languages. This thesis proposes triggered expressions, which supplement the spreadsheet paradigm that enables event driven manipulation of cell events. It also details a practical implementation within spreadsheets, and analyzes some potential use cases.

Thesis Supervisor: Daniel Jackson
Title: Professor of Computer Science
Acknowledgments

It was through great personal turmoil and effort that this thesis was pulled together. This work would not be possible without the guiding presence and support of my thesis advisor, Professor Daniel Jackson. Through his kindness and understanding, he worked with me to make my own journey in academic research. I also want to highlight the knowledge, time, and help from Ph.D. student Matt McCutchinson and postdoc, Shachir Itzhaky. Both provided invaluable technical help along the way through many hours of meetings.

I’d also like to credit Professor Patrick Winston for his enlightening class which I used as guidance to help me write this thesis.

Lastly, I want to thank my parents for their love and support and encouragement which first allowed me to come to MIT.
# Table of Contents

ACKNOWLEDGMENTS .......................................................................................................................... 4  

1 SPREADSHEET PROGRAMMING PARADIGM .................................................................................. 8  
   1.1 WHY SPREADSHEETS ARE POWERFUL ................................................................................. 9  
   1.2 CURRENT EXTENSIONS TO SPREADSHEET FUNCTIONALITY ........................................... 11  
   1.3 AN ALTERNATE DESIGN APPROACH: TRIGGERED EXPRESSIONS .............................. 13  

2 TRIGGERED EXPRESSION SEMANTICS ...................................................................................... 14  
   2.1 SPREADSHEET DESIGN CONCEPTS ............................................................................... 15  
   2.2 FUNDAMENTAL COMPONENTS ...................................................................................... 17  
      2.2.1 Triggers ................................................................................................................. 17  
      2.2.2 Current Value ..................................................................................................... 20  
      2.2.3 Condition ........................................................................................................... 20  
      2.2.4 Formula ............................................................................................................. 21  
   2.3 COMPONENT COMPOSITIONS .......................................................................................... 22  

3 PRACTICAL IMPLEMENTATION OF TRIGGERED EXPRESSIONS ............................................. 23  
   3.1 INFRASTRUCTURE SUPPORT ....................................................................................... 23  
   3.2 PROGRAMMATIC FLOW .................................................................................................. 25  
      3.2.1 Creating a Trigger Expression .............................................................................. 25  
      3.2.2 Event Triggering ................................................................................................... 26  

4 TRIGGERED EXPRESSIONS IN PRACTICE ............................................................................... 27  
   4.1 COMPONENTS .................................................................................................................. 27  
      4.1.1 Counter ................................................................................................................ 28  
      4.1.2 Checkbox .......................................................................................................... 29  
      4.1.3 Radio Button ..................................................................................................... 30  
      4.1.4 Multi-Toggle ...................................................................................................... 31  
      4.1.5 Clock .................................................................................................................. 32  
   4.2 APPLICATIONS ................................................................................................................. 33  
      4.2.1 Chess Clock ....................................................................................................... 33  
      4.2.2 Tic-tac-toe ............................................................................................................ 35  
      4.2.3 To-do List .......................................................................................................... 36  
      4.2.4 Team Collaboration ............................................................................................ 37  
      4.2.5 Habit Tracker ..................................................................................................... 38  

5 DISCUSSION AND NEXT STEPS .................................................................................................. 39  
   5.1 TIME TRIGGER ............................................................................................................. 39  
   5.2 CONSTRAINTS .................................................................................................................. 40  

6 CONTRIBUTIONS .......................................................................................................................... 42  

7 REFERENCES ................................................................................................................................. 43
1 Spreadsheet Programming Paradigm

Spreadsheets present a powerful end-user programming paradigm which allows users to build and customize processes without the need for technical expertise. From the outset of the first spreadsheets application in 1979, the popularity of this visible, functional paradigm encouraged the expansion of new and improved capabilities – initially incorporating charting and database capabilities, and then including ever more elaborate functionality. Fast forward four decades, the norm for spreadsheets includes a range of complex abilities including a set of comprehensive computational functions and data analysis and visualization tools. However, more often than not, each added capability is enabled through added complexity, such as an additional programming language, that breaks the functional paradigm that originally made spreadsheets so popular.

I claim that this need not be so. The core vision of this thesis is that the use of deliberate design can prevent the need for these messy extensions. Here, I propose a design for a whitespace in spreadsheets functionality – dynamic, event-driven manipulation of cells and cell states, which I call “triggered expressions” – that maintains the original functional paradigm. In this thesis, I will demonstrate that triggered expressions obviate the need for some current paradigm breakers and open a whole new set of possible applications within spreadsheets.

Throughout this paper, you’ll be exposed to the context and motivation behind the spreadsheet functional paradigm. You will become familiar with the proposed semantics which enable event-driven, stateful manipulation of cell content. You will explore the potential of the semantics in a prototype and five example applications, and you will be introduced to the implication and
considerations of these semantics. Finally, you will learn about possible future steps that might enable even more powerful applications.

1.1 Why spreadsheets are powerful

Spreadsheets are one of the most prolific end-user programming systems in the world. Current stats mark users of Microsoft Office – including its spreadsheet program, Excel – at an optimistic 1.2 billion users globally ("Microsoft by the Numbers"). Possible applications range from individuals performing simple calculations to large, highly complex systems in businesses such as detailed financial models. In the latter category, spreadsheets have become the backbone for information management in many critical business processes (Baxter 2006). In both cases, the vast majority of users are end-user programmers. End user programmers are programmers who are not professional software developers, yet they might perform some small programming and develop some small processes. This is particularly important because the number of end-user programmers vastly exceeds the number of professional programmers. “In the U.S. alone, the number of end-user programmers is conservatively estimated at 11 million, compared to only 2.75 million other, professional programmers” (Abraham et al. 2009). This means that even though spreadsheets have not been traditionally considered as a programming language, spreadsheets have enabled millions to engage in what is essentially first-order functional programming.

The power of spreadsheets arises from several factors. First it provides a clean yet versatile visual structure for organizing data that is capable of suiting a diverse range of needs. The basic visual structure consists of a two-by-two grid of cells in which users can write values or
formulas. “Computation Models of Spreadsheet-Development” argues that spreadsheets shouldn’t even be seen as an end-user program but instead as “manipulation and presentation of data found in tabular form” and draw a similarity to a calculator (Hodnigg et al. 2004). While I reject the notion that spreadsheets aren’t an end-user-program, the commentary does present the idea that much of the complexity around control and program flow is hidden from the users. Introductory users can slowly construct a program, deriving an intuition much like a child with blocks, building cell by cell. This alludes to a second powerful feature: it provides a transparent view. The continual computation strategy allows the user to see the results on example data as formulas are constructed and modified. At each step, the spreadsheet present a visual feedback system to help direct progress and prevent errors. Users can continue this build and evaluate cycle to formulate increasingly complex processes. Furthermore, spreadsheets use the very same structure and interface used for data to represent the underlying schema and the formulas that define queries on it. Finally, it uses a straightforward and declarative formula language that makes simple data transformations easy and provides support to the user as they build more complex ones.

These features allowed a wide range of users from technical and non-technical backgrounds to create their own applications rather than buying off-the-shelf solutions or paying professional programmers.
1.2 Current extensions to spreadsheet functionality

A key idea in the development of spreadsheets is visual programming. The basic concept makes use of the visual, two-dimensional graphical representations to enable development. In general, this trend has been successful to simplify development, though this has not always resulted in novice users being able to understand how to develop their own applications (Paternò 2013). “End User Development” states that in visual programming, dataflow visual language is a common paradigm that associates icons to high-level functionalities that are important for the specific domain experts. This visual component provided users with a modular higher level interface which was operated as a software building block that implements a general solution and is presented to the user through an interactive frame. Spreadsheets were one of the first major end-user development programming environments enabled by these innovations, beginning with VisiCalc, then continuing with Lotus 1-2-3 and finally, Excel (Paternò 2013).

VisiCalc, the first spreadsheet, supported essential interaction features found in the spreadsheet paradigm: the “What you see is what you get” (WYSIWYG) user interface, ranges with relative and absolute addresses, immediate visual feedback through automatic recalculation, and formula edit through selection of cells and ranges using a mouse (Abraham et al. 2009). The popularity of VisiCalc was in part responsible for the success of the Apple II computer. Lotus 1-2-3 and Borland Quattro were spreadsheet systems developed for the IBM PC platform (Abraham et al. 2009). There are a few commercial spreadsheet systems available today. Among them, Microsoft Excel, now explicitly as a part of the Microsoft Office suite, is the most widely used one. Apple came out with their own twin to Excel, called Numbers. Google’s document suite of applications also includes a spreadsheet system.
Today, spreadsheets have been extended to include more capabilities such as a more powerful programming languages, user-defined functions, and stream processing (Itzhaky et al. 2016). In contemporaries to VisiCalc, users see a range of modern amenities: fancy graphical interfaces, complex computational abilities, and advanced data visualization tools. Users will also find the ability to create automated processes and procedures which can be useful to avoid having to do repetitive, manually intensive tasks or create complex applications. However, these break the traditional spreadsheet paradigm: Excel and other modern spreadsheets use Visual Basic for Applications (VBA), a non-functional programming language that requires a separate interface and obscures sheet functionality.

![Screenshot of a button created in Microsoft Excel which requires VBA code to operate.](image)

In this new paradigm, if users want to create a macro, they must first create a visual concept apart from the cells, such as the button as seen in Figure 1-1, and then select a menu option to modify the behind scenes code that dictates its behavior. Then a dialogue like the one in Figure 1-2 appears, where the user can switch to the VBA programming language. VBA, while a simpler programming languages than others, is not a functional language like the formulas
typically used in spreadsheets. VBA has explicit objects and program flow, typical programming concepts, that now users need to be familiar with in order to make use of these macros. While it provides added functionality, this design approach starts to chip away at the powerful aspects of spreadsheets that made it popular in the first place: a simple and transparent visual interface that is intuitive to use.

![Screenshot of VBA dialog in Microsoft Excel.](image)

**Figure 2-1. Screenshot of VBA dialog in Microsoft Excel.**

1.3 An alternate design approach: triggered expressions

This thesis proposes an alternative to breaking of functional paradigm. Instead of defaulting to imperative programming paradigms such as those used in macros in Excel, what if we design within the existing model? What would such an approach look like? Instead of using VBA, I propose we use the same functional approach to define behavior. However, macros are processes – a series of step by step actions. How could we facilitate that within traditional spreadsheet
notions? I propose we use events to spark a set of consequent changes in cells which might themselves spark more changes. By chaining these changes, we can imitate the same behavior as before. Just as one would do in the scenario described with a macro, a user denotes a cell as a button, clicks it, and that action will drive the manipulation of cell content elsewhere. No more extra dialogue box with a text editor for VBA. Users would then, as a formula within the cell, dictate a more dynamic behavior of content, enabling increasingly complex processes while maintaining the functional programming paradigm of the spreadsheet. Whereas this behavior was previously facilitated through an additional process, called through VBA, we can now operate within the same functional language with the transparent behavior of cell formulas. In this thesis, I call these new, event-driven expressions triggered expressions.

What will these expression look like? What can they really do? In order to facilitate the use of these new expressions, the first step is to delineate a clear set of semantics to be used. These semantics will then be evaluated in the context of a prototype and five example applications.

2 Triggered Expression Semantics

This section discusses the design and considerations of developing semantics of an event driven model for spreadsheets. First and foremost, the design should follow design considerations existing within spreadsheets. Any novel ideas presented within the design should strive to be consistent with the previous semantics. Additionally, spreadsheets are widely used by a wide variety of users and must cater to their needs.
2.1 Spreadsheet design considerations

In light of the fact that spreadsheets are an end-user development tool, it becomes useful to align the design of additional semantics to meet the design considerations of the system as whole. End-User Development (EUD) can be defined as a set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify or extend a software. Again, to clarify, the users here are people who have some basic technological knowledge but are not professional programmers. There are various motivations for EUD: small scale projects don’t warrant the kinds of costs professional development incurs, professional developers lack the domain specific knowledge that end users cannot easily convey, and regular development cycles are too slow to meet the users’ fast changing requirements.

Effective design tools should make it easy for novices to get started (that is, a low threshold) but should also make it possible for experts to work on increasingly sophisticated projects (that is, a high ceiling) (Paternò 2013). The goal of EUD is to allow for a gentle learning curve, which means providing an environment where each incremental increase in the level of customizability only requires the user to apply an incremental amount of effort. Often EUD approaches support users in composing and customizing sets of available basic elements developed by programmers. Some of these criteria can be met or supplemented by specific aspects of the user interface.

In this particular case, the emphasis lies on the semantics meeting both the low threshold and high ceiling. In practical applications, general guidelines around usability can be applied to prompt user and autocomplete formulas for a more comprehensive user-friendliness. In addition,
it becomes pertinent, as an extension to an existing set of semantics and syntax, to consider the alignment and compatibility of the new with the old. The additional syntax should follow the style of the existing formula construction. The semantics should follow the norms of the existing functions, at least where there exists alignable similarities. If there are non-alignable differences, it should be attempted to follow closely related norms, such as in other functional languages.

Therefore, the main considerations for the triggered expression semantics are:

1) Easy to learn
   a. Can a user clearly tell by looking at the triggered expression what the behavior of the cell was likely to do?
   b. Can a user make use of given examples to expand and write their own formulas?

2) Functional
   a. Does it provide expanding functionality for experts to work on increasingly complex projects?
   b. Does incremental functionality require at most incremental construction efforts?

3) Consistent
   a. Does it remain consistent with the functional programming paradigms of spreadsheets?
   b. If not, does it make use of similar concepts as used in other mainstream functional programming paradigms?

Throughout the rest of this section, we will make use of these design consideration to evaluate and support the design decisions made.
2.2 Fundamental Components

The components of the triggered expression semantics consist of four basic aspects: the trigger itself, the condition, the formula, and a current value. The trigger can be defined as the event driving the change in state, which can be a variety of events such as a button or cell click, or a cell content changing value. The condition applies additional logic to control the manipulation. The formula expresses the change which is to happen, and is essentially the same as the traditional notion of a formula with the added ability of using the current cell value. The current cell value dictates the current calculated value of the cell, which introduces the ability to use previous states. In this section, I will detail each of the components, list the set of possible semantics involved in each, and outline the set of possible combinations.

2.2.1 Triggers

The concept of a trigger is the defining novel idea that is introduced in this set of semantics. Triggers are indicated syntactically by the ‘@’ symbol as if to say ‘at’ trigger, the cell content will change. For instance, if you want a cell, called A1, to correspond in some manner to the change of another cell, called A2 – of which its content might be dynamically changing – you could write @A2.change. The behavior of cell A1 is therefore dictated to change each and every time the content in cell A2 changes. This is where the event-driven nature of triggered expression is derived. From this example, we can see that the trigger is essentially an impulse in time, an event such as one might think of in traditional event driven programming. However, for the purposes of spreadsheets, the set of possible events that can be narrowly defined to those in Table 2-1.
<table>
<thead>
<tr>
<th>TYPE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLICK</td>
<td>Click of a pre-defined trigger such as a button</td>
</tr>
<tr>
<td>CHANGE</td>
<td>Change of a pre-defined cell content</td>
</tr>
<tr>
<td>TIME</td>
<td>Change indicated through the passing of time</td>
</tr>
</tbody>
</table>

*Table 2-1. List of possible triggers*

This set of triggers has one key concept in common: each can be defined by an impulse in time. A trigger such as @A1>A2, which might indicate to trigger upon the value of cell A1 being larger than cell A2, would not be feasible. This is because the condition for the event, A1 being greater than A2, would be true for more than an impulse in time. While it might be possible to discretize time into a series of impulses upon which this is triggered, there would need be some infrastructural support for handling an event such as that. This is possible, and in fact is done for time itself, but is not necessary for this case. Instead, it’s possible to use the change trigger and a condition to get the same behavior.

The third case, as mentioned, is a special case in that time is not some visual or schematic component of the spreadsheet. You cannot click on time or manipulate time. It is not represented in the data schema unless intentionally created by the user. Yet, the ability to express behavior as an artifact of time is a critical functionality that should be provided for within the set of semantics. Furthermore, in practice, a user will likely want to define a granularity with which to trigger upon. Therefore, further syntax is required to fully express this.
Another special case that arose, was the ability to trigger on some number of triggers. Because an event is represented as an impulse, the only possible outcome is the OR of two or more triggers. AND becomes impossible as matter of timing – even if the events appear simultaneous to a human, there will be an infinitesimal difference in the practical mechanics of it which is further complicated by being nondeterministic. Neither is ordering, the equivalent of @trigger1 then trigger2, a possibility. To do so would require the maintenance of a complete history of all trigger events or some alternative hack-ish approach. For the purpose of simplicity, therefore, the only computation on triggers is the OR; in English, this would be the equivalent of “if trigger1 happens or if trigger2 happens, then..” To accommodate this functionality, the addition of a keyword, any, was introduced.

Therefore the full set of syntax for triggers is as follows:

\[
\text{List 2-1. BNF for triggers grammar.}
\]

```plaintext
<trigger-object> ::= <cell> | <button>
<trigger-click> ::= "@" <trigger-object>
<trigger-chang> ::= "@" <trigger-object> ".change"
<trigger-time> ::= "@Time.second(" <number> ")"
<trigger-any> ::= "@any(" <trigger-set> ")"
<trigger-set> ::= <trigger> | <trigger><trigger-set>
<trigger> ::= <trigger-click> | <trigger-chang> | <trigger-time> | <trigger-any>
```
2.2.2 Current Value

Now that the event driven aspect of triggered expressions has be established through the notion of triggers, we must evaluate how a cell content might be manipulated given that event. In order to allow for functionality beyond the simple notion of “setting” and “resetting” cell values, the spreadsheet must keep track of the current value of the cell. This would allow for users to do something as simple as implement a counter: given a trigger, increment the value of the cell by 1, or increasingly complex logic such as given a trigger, take the current value, perform some computation and evaluate to a value based on the result. A notion of a current value also provides the opportunity for users to set a base case or starting value and interject a specific value if desired. Finally, if a cell with a triggered expression is referenced elsewhere, the current value will be the value used. The representation for current value is indicated by “this” or by the cell name as one might do in any existing spreadsheet application.

2.2.3 Condition

Conditions are a largely unchanged concept. The purpose of including conditions into the set of possible semantics is to provide finer granular control over the situations under which a user might want to modify cell content. Perhaps, for instance, if a user wanted a cell to stop incrementing when it reached a max limit, a condition could be used to check for the current value of the cell by writing ‘if(this == maxValue)’. As in current spreadsheets, conditions can include a wide range of stipulations, but boil down to two simple cases:

- To check a computational result
- To compare a cell value to some other value
The one additional notion that is included in this set of semantics is that conditions can now include statements made on the current value.

Therefore the full set of syntax for conditions is as follows:

<Boolean-expression> ::= <math-computation> | <cell-comparison>
<statement> ::= "" | <condition> | <formula>
<condition-statement> ::= "if(" <Boolean-expression> ")"
<statement> "::" <statement>

List 2-2. BNF for condition statements grammar.

2.2.4 Formula

The formula, as with the concept of the condition, is almost fully aligned with the traditional notion of formulas in spreadsheets. This concept is the expression to be evaluated upon successful triggering and affirmed conditions. It is the defining behavior of the cell content, and is in essence, the same formulas seen in existing spreadsheet applications. These include, but are not limited to, performing calculations and setting or resetting cell content. As with conditions, the novel aspect to this component is the ability to perform calculations and manipulations on the cell current value as indicated by ‘this’ or the cell name.
2.3 Component compositions

Now that each component has been introduced, we can explore the variety of ways in which they can be combined into full triggered expressions. The general order of a triggered expression is first the current value, followed by the trigger, condition statement which includes a formula:

\[
<\text{composition}> ::= <\text{current-value}> \text{ "|" } <\text{trigger}> \text{ "|" } <\text{condition-statement}>
\]

The sole required component to be a triggered expression is, in fact, just the trigger. Each component can then optionally be included to form ever increasing complex functionality. Additionally, triggered expressions can be chained. This allows for a more comprehensive notion of OR-ing triggers which can now include differing behavior dependent upon which trigger is triggered.

Therefore the full set of possible combinations is as follows:

\[
<\text{triggered-expression}> ::= <\text{trigger}>
\quad | \\
<\text{trigger}> \text{ "|" } <\text{formula}>
\quad | \\
<\text{trigger}> \text{ "|" } <\text{condition-statement}>
\quad | \\
<\text{current-value}>,<\text{trigger}> \text{ "|" } <\text{formula}>
\quad | \\
<\text{current-value}>,<\text{trigger}> \text{ "|" } <\text{condition-statement}>
\]

*List 2-3. BNF for triggered expression grammar.*
3 Practical Implementation of Triggered Expressions

Through an implementation, we can evaluate the design decisions made, uncover unanticipated problems, and expose implications of the design. The following subsections will cover what additional infrastructure was required to support triggered expression, program flow for evaluating triggered expressions, and discuss the considerations and implications of the practical implementation.

3.1 Infrastructure support

The prototype implementation is based on a set of development technologies – meteor as a web development platform and its related packages. Meteor was chosen for several reasons. First, Meteor is a free, open source web framework that facilitates rapid prototyping. Meteor handles all the messy details of initial setup and provides the basic foundations to get things going. Second, Meteor is a full stack JavaScript framework. JavaScript was a familiar language and has all kinds of functionality regarding events and event handling – useful for event-driven expression. Lastly Meteor has built and gathered a collection of libraries and packages. Again, why reinvent the wheel. The purpose of this part of the project was not to develop a fully functional application, from the ground up, but to demonstrate how a spreadsheet might incorporate triggered expressions. Therefore, the use of Meteor and its packages allowed the vast majority of effort to be directed at the implementation specific to triggered expressions.

In particular, there was one package, handsontable-meteor, which incorporated the latest Handsontable. Handsontable is a comprehensive spreadsheet library for developers. It
incorporates a wide range of features, as is visible in Figure 3-1, seen commonly in modern spreadsheets today. Handsontable is also lightweight with little overhead required and is easily extensible. The API provided useful hooks on over a 100 spreadsheet specific events such as beforeValueRender or afterOnCellMouseDown. This suit the exact needs for this thesis project.

<table>
<thead>
<tr>
<th>ID</th>
<th>Country</th>
<th>Code</th>
<th>Currency</th>
<th>Level</th>
<th>Units</th>
<th>Date</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EUR</td>
<td>EUR</td>
<td>EUR / USD</td>
<td>0.9033</td>
<td>08/19/2015</td>
<td>-0.20%</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>JPY</td>
<td>JPY</td>
<td>JPY / USD</td>
<td>324.3870</td>
<td>08/19/2015</td>
<td>-0.01%</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>GBP</td>
<td>GBP</td>
<td>GBP / USD</td>
<td>0.6396</td>
<td>08/19/2015</td>
<td>-0.03%</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>CHF</td>
<td>CHF</td>
<td>CHF / USD</td>
<td>0.9775</td>
<td>08/19/2015</td>
<td>-0.08%</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>CAD</td>
<td>CAD</td>
<td>CAD / USD</td>
<td>1.3097</td>
<td>08/19/2015</td>
<td>-0.05%</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>AUD</td>
<td>AUD</td>
<td>AUD / USD</td>
<td>1.3588</td>
<td>08/19/2015</td>
<td>-0.20%</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>NZD</td>
<td>NZD</td>
<td>NZD / USD</td>
<td>1.3218</td>
<td>08/19/2015</td>
<td>-0.36%</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>SEK</td>
<td>SEK</td>
<td>SEK / USD</td>
<td>8.5280</td>
<td>08/19/2015</td>
<td>-0.16%</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>NOK</td>
<td>NOK</td>
<td>NOK / USD</td>
<td>8.2433</td>
<td>08/19/2015</td>
<td>-0.08%</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>BRL</td>
<td>BRL</td>
<td>BRL / USD</td>
<td>3.4860</td>
<td>08/19/2015</td>
<td>-0.09%</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>CNY</td>
<td>CNY</td>
<td>CNY / USD</td>
<td>6.3961</td>
<td>08/19/2015</td>
<td>-0.04%</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>RUB</td>
<td>RUB</td>
<td>RUB / USD</td>
<td>65.5980</td>
<td>08/19/2015</td>
<td>-0.59%</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>INR</td>
<td>INR</td>
<td>INR / USD</td>
<td>65.3724</td>
<td>08/19/2015</td>
<td>-0.26%</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>TRY</td>
<td>TRY</td>
<td>TRY / USD</td>
<td>2.8689</td>
<td>08/19/2015</td>
<td>-0.92%</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>THB</td>
<td>THB</td>
<td>THB / USD</td>
<td>35.5029</td>
<td>08/19/2015</td>
<td>-0.44%</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>IDR</td>
<td>IDR</td>
<td>IDR / USD</td>
<td>13.8300</td>
<td>08/19/2015</td>
<td>-0.09%</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>MYR</td>
<td>MYR</td>
<td>MYR / USD</td>
<td>4.0945</td>
<td>08/19/2015</td>
<td>-0.10%</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>MXN</td>
<td>MXN</td>
<td>MXN / USD</td>
<td>16.4305</td>
<td>08/19/2015</td>
<td>-0.17%</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 3-1. Screenshot of Handsontable*

However, beyond the specific technologies of the prototype, an implementation of triggered expression requires additional application support. Spreadsheets that use triggered expressions will need the addition of two new components: a handler register and a triggered expression parser, and the modifications of another two processes, rendering and referencing. The first new component, the trigger expression parser, is used to validate the expression. An incorrect expression, in the current prototype, will result in no action as if a user entered a string. The second new component, a handler register, creates and manages the appropriate event handlers for each new trigger. These event handlers can then be called by the corresponding trigger action. The handler register also verifies that the trigger exists, otherwise no event handler will be created, and the special case of the Time trigger. Spreadsheet rendering now must check if a cell contains a triggered expression and render the current value. This is the case for formulas, =
1+1 will render to a 2, and is extended to include the triggered expressions. Referencing a cell with a triggered expression will also need to return the current value of a cell.

3.2 Programmatic Flow

This section details the process flow of the infrastructural support. Specifically, it handles two processes, the creation of triggered expressions and its corresponding behind the scenes setup and the triggering of an event and its corresponding change to the cell value. In its current state, the prototype follows a naive approach; optimizations will be left for future steps.

3.2.1 Creating a Trigger Expression

![Program flow for the creation of a triggered expression.](image)

*Figure 3-2. Program flow for the creation of a triggered expression.*
After any cell is edited, a hook on afterChange calls the parser, handing off the text content of the cell. The parser attempts to match the trigger pattern of ‘@’ followed by non-space characters. It then looks for the corresponding formula or a combination of a condition and formula. The parser returns a JSON object, called a trigger object. If the parser cannot find any triggers or condition and formula pairs, an empty list is returned. After receiving a trigger object, the hook checks for a non-empty list of triggers; if so, it calls the event handler, passing on the trigger object. The event handler checks first if any of the triggers are the special case for time, parsing the time units and creating a corresponding function `window.setInterval()`. If the trigger is not a special case, the event handler checks to verify the trigger or cell exists. If not, an alert is fired to call user attention to the error. If the trigger does exist, the event handler register creates an event handler for the event, either a click or change, on that trigger DOM object.

3.2.2 Event Triggering

![Figure 3-2. Program flow for the firing of a trigger.](image)
If a trigger is triggered, the event handler, which was established upon the triggered expression creation, fires. The event handler calls render on the spreadsheet with the trigger name as a parameter. Render looks through every cell, parses the cell content, and looks to see if a cell has a triggered expression with the appropriate trigger name. If a matching triggered expression is found, the render function grabs the current value and saves it into a variable to be evaluated in the case any formula or condition mentions ‘this’. Next, the conditions are evaluated, if any exist. Upon an affirmative evaluation of the conditions, the corresponding formula is evaluated. The render function saves the result and correctly modifies the cell content to reflect the new current value. Finally, the render function sets the display value as the current value so the cell appears to the user as it would for a traditional function.

4 Triggered Expressions in Practice

What can triggered expressions do? What additional power do they provide over existing spreadsheet applications? The answers to these questions will be explored in the following sections. First, we’ll evaluate what additional and reusable components can be created with the use of triggered expressions that was previously impossible, and then we’ll look at potential applications that might use those kinds of components and other powerful uses of triggered expressions.

4.1 Components

Components can be defined as a set of features or functionality, combined into one widget or component, that can be reused in the creation of an application. This is typically a terminology used in web application development; however, triggered expressions facilitate the use of these
modular components inside spreadsheets. This is extremely powerful – it allows for automation and dynamism with little user effort and without needing additional programming languages. This section covers several commonly used components in applications, and discusses how a user might develop them with the semantics of triggered expressions.

4.1.1 Counter

A counter is a simple incrementer, literally used to count the occurrences of some action. Useful applications of a counter include: tracking number of changes to a dataset, tracking the number of turns, or the number of items deleted. With triggered expressions, creating a counter is straightforward. The template for a counter would consist of a trigger, the event upon which the user wishes to count, and then a simple formula: this + 1. For example, in order to create a counter for keeping track of number of turns, the triggered expression might look like this:

```plaintext
@turnButton | if(A1 == 'blue') this+1 :
```

The idea would be that there exists some cell designated as a turn button to indicate the end of a turn, say between a red and blue team. Then, upon the clicking of the button, the counter would check to see if it was the correct team’s turn, and increment itself.

This template could be used for increasingly complex behavior. The counter could increment under very specific behavior, using conditions to make comparisons or evaluate cell content. Additionally, the counter could increment on any number of triggers available. For instance, in a program could track a series of tasks that have a status cell. Upon any one of the status cell being completed, the counter could increment.
4.1.2 Checkbox

Another common component is a checkbox, toggling between checked and unchecked. Checkboxes are typically used to select a set of items or to indicate completion of a task. In triggered expressions, this translates to a toggling of current values, from checked to unchecked. The template for the triggered expression would be something like:

```plaintext
unchecked, @this | if(this == 'unchecked') 'checked' :
 'unchecked'
```

Like the counter, this behavior could be make increasingly complex depending on the wishes of the user. For instance, one could want another trigger elsewhere, say a check-all button, to cause the cell to become ‘checked.’ One note here is that the user has to initialize the current value with the appropriate ‘checked’ or ‘unchecked’ value, and if the current value is changed to some inappropriate value for the triggered expression, there will be abnormal behavior. As the template is set above, anything over than ‘unchecked’ will default the value of the cell to ‘unchecked.’ This is a nice trick to catch any undeliberate changes that might throw off the correct behavior. Therefore, upon an error, the next trigger will cause the cell value to change back to ‘unchecked’ and the proceeding trigger will continue with correct behavior. This may cause the checkbox to become asynchronous with other changes – such as upon a check-all button – so the user will still need to be careful.
4.1.3 Radio Button

The radio button is the next step up from the checkbox. A radio button is also used to select a set of items; however, it has an added constraint that only a certain number of items may be selected at any one time. Some instances of radio buttons allow up to some n number of item to be selected, preventing any more to be selected or randomly choose a previously selected item to be unselected. At the micro-level, the cell will have the same toggling of ‘unchecked’ to ‘checked.’

The additional constraints make the creation of radio buttons tricky. In order to implement a radio button through triggered expressions, a user will need to create additional set up not required in checkbox creation. First, a user will need to define the ‘n’ which is the constraint for the number of items allowed to be selected. For the sake of the example, let’s set this ‘n’ to be 2. For a more modular use, the best case would be to set this value in a cell. In this example, we’ll just use the number to be more straightforward. Second, the user will need to create a cell that contains a count of the number selected – this will allow for the radio buttons to check whether they can be selected. The counter, as described above, will be a somewhat more complex counter triggered expression:

\[ B1: \text{@any}\{A1, A2, A3\} | \text{sum}(A1, A2, A3) \]

This expression assumes that the radio buttons are in cells A1 – A3, and that instead of toggling current value between ‘checked’ and ‘unchecked,’ there’s a toggle between the value of 0 and 1 where 1 is selected and 0 isn’t. The behavior of the cell is then, on any change of the radio buttons, re-sum the values of each to get an updated count of the number of selected buttons.

Therefore, the triggered expression for the radio button would look like:

\[ A1: 0, \text{@this} | \text{if}(this == 0 \& B1 < 2) 1 : 0 \]
Therefore, the radio button will check first to see that the current value is the appropriate value, 0 or unchecked, and then checks to make sure it can be selected, i.e. that the cell keeping track of the count of selected buttons is less than the ‘n,’ 2.

While this may seem particularly hack-ish and have a not insignificant overhead, the alternative approach in currently available spreadsheets is non-existent. Furthermore, simple copying and pasting can be used to expand the number of radio buttons or to create multiple sets. This is a minor effort compared to the increased functionality available.

4.1.4 Multi-Toggle

Another extension of the checkbox concept is a toggle between multiple values. This can be used to track the turns of multiple players, indicate status progression of a task, or increasing sizes. Like the checkbox, this is a simple change based on the current value. The template for the triggered expression might look like:

'a', @this | if(this == a) 'b' : (if (this == 'b') 'c' : 'a')

Therefore, on click, the value of the cell will change from ‘a’ to ‘b’ to ‘c’ and back to ‘a.’ Like with the checkbox, there is a default behavior here of resetting to ‘a.’ Additionally, as with the checkbox, the template can be expanded to facilitate increasingly complex behavior such as having multiple triggers or more specific conditions. This pattern can be expanded to include any number of multiple options. The triggered expression does expand linearly with the number of options included, and can become quite unwieldy with a set larger than 4. However, as mentioned in the discussion in the design section, this is also true with modern spreadsheets.
4.1.5 Clock

One particularly exciting component that is now possible with triggered expression is the notion of a clock, or really any extension of this such as an alarm, stopwatch or timer. This was in fact the sole purpose of including the special case trigger for time because it allows the exploitation of time in this way. Uses could include timing a player’s turn or tracking how long a task is left outstanding. The template for a traditional clock would look like the following:

```excel
@Time.second(1) | NOW()
```

Where ‘NOW()’ is a common function in Excel that returns the current date and time. A template for creating an alarm would look something like:

```excel
@Time.second(1) | if(NOW() == alarmTime) ‘ALARM!’:‘not yet’
```

Where alarmTime would be some specified date and time. Additionally, the user can set a conditional formatting on the cell to be visually obvious upon displaying ‘ALARM!’ as the current value. The use of ‘NOW(),’ specified as a date and time, would allow a user to set multiple timers for specific days, but has the side effect of making it more difficult to set repeating, or day agnostic, alarms. A triggered expression for a timer might look like:

```excel
endTime, @Time.second(1) | if(this > 0) this-1 : 0
```

Where endTime is some amount of time, in this case determined in seconds, and at each interval of a second, one second is reduced from the current value. A user can also place a conditional formatting rule on the cell to visually highlight when the timer is at 0. Finally, to write a triggered expression for a stopwatch, a user might type something along the following lines:

```excel
0, @Time.second(1) | this+1
```
4.2 Applications

With the new toolset of components enabled through the use of triggered expressions, discussed above, we can look at a number of new applications that are now possible. Through the implementation of these applications, I demonstrate the power of triggered expressions and draw implications of their use.

4.2.1 Chess Clock

The traditional chess clock consists of two count down timers, one for each player. The clocks track the set amount of time each player is given in which they are allowed to think and make moves in a game. The clocks are linked in such a way that only one clock ticks down at a time. A turn is concluded by that player hitting a button on the top of the clock which pauses their timer and starts their opponent’s timer. The player with the black pieces starts first. There are several modes of play, each with different tracking of time. For this example, we will “Rapidplay” in which each player is given a total time of 30 minutes to make all the moves needed in a game.

To represent this contraption inside a spreadsheet with triggered expressions enable, a user would need to set up three different cells: one for each clock and one to track whose turn it is. The triggered expressions would look like the following:

```
A1 - clockBlack: 1800, @Time.second(1) | if(this > 0 & C1 == 'black') this-1: this
```
B1 - clockWhite: 1800, @Time.second(1) | if(this > 0 & C1 == 'white') this-1: this

C1 - Turns: ‘black’, @this | if(this == ‘black’) ‘white’ : ‘black’

The two clocks, designated cells A1 and B1, make use of the template for the timer as described in 4.1.5. with the added conditional check to ensure that the timer only counts down when it’s the correct player’s turn, designated by cell C1. C1 makes use of the checkbox component as described in section 4.1.2. only now switching between the two players. The trigger for cell C1 is indeed itself; therefore, players will indicate that their turn is complete by clicking on the cell.

A less concise implementation, however more accurate to the mental model of a chess clock, would be to add two additional cells – let’s call them A2 and B2 – to represent the buttons that each player would push in real life to indicate the end of a turn. The new triggered expressions would look like so:

C1: ‘black’: @A2 | ‘white’, @B2 | ‘black’

This expression makes use of chaining two triggers, now the two buttons for ending turns. A fun trick occurs here, where the checking of conditions – i.e. to make sure that the player clicking the button is actually the player whose turn it is – because the in the case that it is the player’s turn, it should switch to the other player, and in the case it isn’t, it should stay the other player’s turn. In concrete turns, if the black player correctly has the turn and clicks A2, C1 should correctly toggle to the white player’s turn. If the black player does not have the turn, meaning it is white’s turn, and incorrectly clicks A2, then C1 should maintain that it’s the white player’s turn. The converse is true should it be the white player clicking B2.
4.2.2 Tic-tac-toe

Another interesting game that can be implemented with triggered expressions is tic-tac-toe. The childhood game consists of two players, represented by ‘X’ and ‘O’, who wish to get three of a kind in a row in a three-by-three grid. The players switch off marking a square in the grid with their respective characters.

In a spreadsheet with enabled triggered expressions, a user would need 10 cells: 9 for the three-by-three grid and one to indicate whose turn it is. The expressions would look like the following:

```
Grid cell: @this | A4
A4 - Turn: 'X' @any({gridCell.change}) | if(this == 'X') 'O': 'X'
```

In this case, the cell that indicates the turns, A4, would directly indicate whose current turn it is through the template for a checkbox. It also makes use of the ‘any()’ paradigm to indicate that on the change, or write in, of any grid cell, it will trigger a change of turn. The cell that is clicked would then use the value of A4 to write in the player’s character. Of note, the ‘.change’ is important to imply an ordering on the events. If a player clicked a grid cell, the spreadsheet would run the nondeterministic process of selecting whether to evaluate cell A4 or the cell clicked first. This could lead the situation where A4 is switched to indicate a new turn and then the cell clicked is filled in with the new turn, not the current one.
4.2.3 To-do List

Moving away from game applications into productivity, a common application is some task tracker or to-do list. It would consist of a list of tasks with associated names and statuses. A user would then add, delete, and complete tasks as needed.

A simple triggered expression implementation would look like the following:

```
taskStatus: 'in progress', @this | if(this == 'in progress')
              'complete' : 'in progress'
```

For each status, separated from the task name, a checkbox component would be used to toggle back between complete or in progress. To increase the complexity, say by adding a delete option, there’s two options: first, to add two cells to operate as buttons to indicate complete or delete and chain both triggers, or second, use a multi-toggle to be able to toggle between the three states. This would look like:

1. `taskStatus: 'in progress', @completeBtn | if(this == 'in progress') 'complete' : 'in progress', @deleteBtn | 'delete'

2. `taskStatus: 'in progress', @this | if(this == 'in progress') 'complete' : (if(this == 'complete') 'delete': 'in progress'

Additionally, one might want a separate button to apply to all tasks such as complete all or delete all. This would occur by chaining the above triggered expression with:

```
@deleteAll | 'delete', @completeAll | 'complete'
```
Finally, a user may want to associate some deadline with a task. This would need to be a
separate cell from the task status. It would use an alarm template and might look something like:

```plaintext
@Time.second(60) | if (NOW() == endTime) 'ALARM!': endTime -
                      NOW()
```

4.2.4 Team Collaboration

Another interesting productivity application is one to administer collaborative teams, which
might include tracking each member’s status and handle tasking. This is a useful application for
product managers or more informal groups who need to manage tasks across a set of people.

In a spreadsheet, a user could create this by having a master list of team members with their
status, maybe “busy” or “available”. Separately would be another list of tasks, again with a
status, due date, and person assigned to the task. The triggered expressions might look like:

```plaintext
taskStatus: 'incomplete', @this | if(this == 'incomplete') 'in
       progress' : (if(this == 'in progress') 'complete' :
       'incomplete')

taskAssignee: @taskStatus | if(this == '' & tastStatus == 'in
       progress') 'who’s working on this?': this +'busy'

taskDate: @Time.second(60) | if (NOW() == endTime) 'ALARM!':
       'time left: ' + endTime - NOW()
```
Just as before, the task status is a multi-toggle component which will follow the progression of “incomplete” to “in progress” to “complete” upon clicking the status cell. Then, as the status of a task changes, the cell that tracks who’s assigned to the task gets modified to indicate that that person is busy, if there is someone assigned. The master list of team members and their status gets updated every time an assigned task changes, using an Excel function to evaluate if there are any tasks that have an assigned person which has a task in progress.

4.2.5 Habit Tracker

The last example application built is another productivity app. Recently, daily journaling and habit tracking have become a popular way to monitor personal productivity and introduce new habits into a lifestyle. While some people prefer to have a physical journal, oftentimes, people will use an online spreadsheet like Excel to more easily track their habits. The spreadsheet would contain a list of habits running down the rows and a list of days across the columns. Then a user would put a 1 or a 0 to indicate either a successful completion of a habit that day or not. This would be a tedious process for more than just a handful of habits each day, which is contrary to the goal of habit tracking. It should be a short, less than 30 second activity, that can be done daily.
With triggered expressions at play, a habit tracker might look something like this:

```plaintext
dayHabit: '0', @this | if(this == '0') '1' : '0', @currentDay | '1'
```

This simple formula would allow a user to click through the habits they’ve completed for the day with the checkbox component paradigm. Alternatively, if the user has completed the majority of habits for that day, they can click on the cell for the date, which will autofill all the habits as complete or ‘1’. Then the user could go back and uncheck the habits that they didn’t complete – a more efficient process, especially when habits typically reach 30+ activities.

## 5 Discussion and Next Steps

After building the implementation and looking at a few sample applications that can be built with triggered expressions, there’s a few points of discussion that arise.

### 5.1 Time Trigger

First is time. In actually implementing the special trigger for time, it became clear that performance was an issue. As described in section 3.1., the process for handling a trigger naively iterated through the entire sheet and looked for a cell that contained a trigger expression with time. While this process handled fine with a click or change trigger, this was not the case for time which was by definition, time dependent. In particularly, if there was a trigger set on ‘Time.second(1),’ the process was not fast enough to look at all the cells and analyze its contents.
for a correct triggered expression and evaluate it. As a workaround in building the applications, time was set to be triggered at a larger interval, typically a minute.

Additionally, the trigger for time has the issue of never ending. While in Excel Macros there’s an explicit instruction in VBA to stop a timer, this is not so with a trigger. This idea, that a Time trigger should stop being valid at some point, is contrary to the behavior of the other triggers. One available option was to quit the spreadsheet application, or in the prototype case to suspend the meteor application. However, upon opening the application up again, the trigger would begin anew. This might lead a user to wanting to “pause” the triggered expression for some time, say to reset it or modify the behavior without the current value being modified every so often. In the existing implementation, a user would do this by modifying the triggered expression to essentially do nothing, i.e. have no formula. Then a user can modify the spreadsheet as they see fit and “restart” the time trigger by going back to reinstate the correct formula. Again, this is troublesome to the user as they would have to clear out all triggered expressions with a time trigger in it. This leads to the need for a specialized case, a stop trigger. The user would need to declare a cell as the stop trigger, and upon clicking it, it would short circuit triggered expressions with time. In code, this would be a simple Boolean that would be checked before following the program flow for calling a re-evaluation upon a time trigger.

5.2 Constraints

With triggered expressions, a critical issue with its functionality is that user error can drastically change the behavior of the spreadsheet. For instance, a user can set the current value to essentially any current value they see fit. There is no constraint on the current value of the cell to
fit within the confines of what the triggered expression knows what to do with. As an example, we can look at the Team Collaboration example application. If a user misspells a name when assigning people to tasks, the list of team members will incorrectly relate that the member is available. Additionally, being able to modify the current cell value of the triggered expression seems contradictory to correct behavior to begin with – shouldn’t the current value of the cell only be the result of a correct evaluation of the triggered expression?

While it may appear that these are issues with triggered expressions, these are in fact issues with the underlying flexible structure of spreadsheets. The amount of user error in spreadsheets is incredible. Various studies indicate that upwards of 88% of spreadsheet applications have errors (Olshan 2013). The real issue? Humans. Yet other applications find ways to minimize human error. The traditional spreadsheet is extremely unstructured, defined by a two-by-two grid with loose, adhoc formulas. Any “real”, or semantic structure is defined solely in the minds of the user, which leaves it open to large rates of user error and makes it difficult to hand off to other users. This has engendered efforts to create a new infrastructure for spreadsheets to combat this. An example is Object Spreadsheets. Object Spreadsheets provides support for nested, variable-sized lists and object references. Its computational model treats the data as a relational table instead of a loosely defined matrix. This, with the ability to reference objects and object properties, creates the infrastructure to back the semantic model of a user. An interesting next step would be to implement triggered expressions within the context of Object Spreadsheets.
6 Contributions

In this thesis, I presented an alternative design approach to adding complexity through more complexity. In the case of spreadsheets, an end-user program, ensuring the design of the application is easy to learn, functional, and consistent is critical to its widespread applicability. I propose triggered expressions as a functional paradigm for event driven manipulation of cell states which maintains consistency with the original functional paradigm of the spreadsheet. Through triggered expressions, users can express an expanded set of functionalities without defaulting to imperative programming languages. I also developed an example implementation of triggered expressions within spreadsheets, discussed some potential use cases, and explore implications of its use.

In summary, I made the following contributions:

1) established the context and motivation behind the spreadsheet functional paradigm
2) explored a set of new semantics to include stateful manipulation of cell content
3) demonstrated the potential of the semantics in a prototype through five examples
4) discussed the implication and considerations of the semantics given the prototype implementation
5) outlined future steps including performance optimization and the use of alternative
7 References


https://pdfs.semanticscholar.org/e2b5/2b331a87762ae7f3e0cfeff8a7669281891b.pdf


<https://news.microsoft.com/bythenumbers/>
