```
-- This is a comment

#title What is the best tool for data visualisation?

#type BarChart

#data
    Marker      99
    Excel       32
    Google      62
    Infogram    18
    Highcharts  10
    Doodles     12

#background pink

@highest
    label Awesome

@lowest
    label Bad
    color red
```

Data visualisation with a simple syntax

DATX02-15-04

Jimmy Hedström, John Hult, Patrik Göthe,
Oscar Nilsson, Morhaf Alaraj, David Nääs

Institution for Computer Science and Engineering
Supervisor: Bengt Nordström

Chalmers University of Technology
Gothenburg, 2015

Foreword

We would like to thank our mentor Bengt Nordström for the time spent helping us. His approach to both the subject, and academic writing, was invaluable.

Jimmy Hedström, John Hult, Patrik Göthe, Oscar Nilsson, Morhaf Alaraj, David Nääs, Gothenburg, 2015-06-02.

Abstract

This thesis describes the development of *Marker*, a syntax based data visualisation platform with the power and flexibility of a programmers tool, but usable by anyone, programmer or not. Today more open data than ever is available thanks to the internet but few people have access to the means to use it. Consequently, some actors on the internet have the ability to analyse and draw conclusions from data while some do not. *Marker's* purpose is to bridge the gap that this creates.

The project was divided in two parts, a pre-study and the actual development of the application. The pre-study was conducted as a literature study and in the form of interviews with experts on the area of data visualisation. The development was documented in the form of system design decisions and a full description of the end result.

Based on the results, *Marker's* usefulness and viability in comparison to similar tools and its intended purpose is discussed. The conclusion was made that *Marker's* usability and extendable system design adds new value to the area of data visualisation. However, more development and user testing is required in order to make *Marker* a competitive alternative to the existing data visualisation tools of today.

Sammandrag

Denna rapport beskriver utvecklingen av *Marker*, en syntaxbaserad plattform för datavisualisering, lika kraftfull och flexibel som ett programmeringsverktyg men ämnat för att användas av vem som helst. Tack vare internet, är mer öppen data än någonsin tillgänglig, men få personer har tillgång till den kunskap och de verktyg som krävs för att kunna dra nytta av situationen. Följaktligen kan vissa aktörer på internet analysera och dra slutsatser från data medan andra inte har den möjligheten. *Markers* syfte är att överbrygga klyftan som detta skapar.

Projektet delades in i två delar, en förstudie samt den faktiska utvecklingen av applikationen. Förstudien utfördes genom en litteraturstudie samt intervjuer med sakkunniga inom datavisualisering. Utvecklingen dokumenterades i form av designval av systemet samt en fullständig beskrivning av slutresultatet.

Baserat på resultatet diskuteras *Markers* relevans och nytta i förhållande till liknande mjukvara samt projektets syfte. Sammanfattningsvis bidrar *Markers* enkelhet och utbyggbara design med nya värdefulla egenskaper till datavisualiseringsområdet. Trots detta krävs vidare utveckling samt användartester för att göra *Marker* till ett konkurrenskraftigt alternativ.

Glossary

API: An Application Programming Interface is defined rules about how a program can be communicated with from other programs.

Atom: A core building block of the *Marker* system, capable of turning JSON-formatted data into a specific graph or chart.

CSS: Cascading Style Sheets is a computer language used for describing the look and formatting of a document written in a markup language.

Framework: In computer systems, a framework is often a layered structure indicating what kind of programs can or should be built and how they would interrelate. Some computer system frameworks also include actual programs, specify programming interfaces, or offer programming tools for using the frameworks.

GUI: Graphical User Interface. An interface of a computer program that allows users to interact with the system by visual means.

HTML: HyperText Markup Language is the standard markup language used to create web pages. A markup language categorises content into predefined elements.

JavaScript: A programming language commonly used to control and manipulate data in a web browser.

JSON: JavaScript Object Notation, a data-interchange format. A way to define and describe objects in JavaScript.

Marker: The name of the data visualisation tool developed in this project.

Molecule: A central hub in the *Marker* system, keeping track of all available *atoms*.

Parser: Parsing or syntactic analysis is the process of analysing a string of symbols, either in natural language or in computer languages, conforming to the rules of a formal grammar.

Platform: Short for *Computing Platform.* A computing platform is an environment (hardware of software), designed for running other applications. A computing platform defines a set of rules that applications adhere to, which make the application able to run within the platform. *Marker* is a software based platform for applications that visualise data.

SVG: *Scalable Vector Graphics.* A markup Language for creating vector graphics. *SVG* is used in *Marker* to draw the components of a graph.

Syntax: In linguistics, syntax is the set of rules, principles, and processes that govern the structure of sentences in a given language.

Contents

1 Introduction

During the last decade, the internet has become a significant part of our lives. For example, the average Swedish person spent more than 21 hours online per week during 2014 (Findahl, 2014). In addition to this, data from behaviour on popular sites like *Facebook* is now continuously tracked and stored (Felix, 2012). Most of this data is kept private by the companies but more and more data from sources that were previously inaccessible is being made public through initiatives like *Open Knowledge* ("Open Knowledge," 2015) and *European Union Open Data Portal* ("European Union Open Data Portal," 2015).

1.1 Background

When using data to ones advantage, valuable knowledge can be extracted. However, access to this knowledge is restricted to those who know how to extract it from the data. Individuals, organisations and enterprises can gain advantages in decision making if they manage to access this knowledge. One way to accomplish this is through visualisations that converts data into graphics, which are more comprehensible to the human mind. However, creating such visualisations often require some form of knowledge in programming. Actors possessing such skills are therefore in an advantageous position.

Today, user friendly tools exist that are used for formatting text. *Markdown*, for example, is a simple and extremely popular syntax for formatting and working with text in order to compile it into *HTML*. *Markdown* is regarded as effective in many niches, supposedly because of its simplicity and flexibility which makes it easy to learn. It is easy to teach to a non-programmer. If the user makes a syntactical mistake the program will not crash any processes when being compiled or parsed, instead the program will not render the correct formatting ("Getting the gist of Markdown's formatting syntax," 2015). *Markdown* is used in many applications for formatting text. One example of this is the large online community Reddit, this is demonstrated in appendix A1.

This suggests that it is possible to construct a syntax that is simple enough for a non-programmer to learn yet powerful enough to generate adequate visual results. The syntax must be defined by a creator, with defined rules about how it works, which then may be interpreted by a parsing program that reads the syntax and converts it to a data format usable by an application. While this is a proven method for text formatting with *Markdown*, it might also be useful for data visualisation purposes.

1.2 Problem statement

While the amount of data available to the general public is larger than ever, few people have the ability to take advantage of this situation. This is due to the fact that the majority do not have experience in programming and possess little or no knowledge about how the internet actually works theoretically. This gap between users is a problem since it creates hierarchy and consequently inequality.

This problem could be seen as an extension to the *digital divide* debate which concerns economic and social inequality and access to information and communication technologies, i.e. the internet. However, equality on the internet is not only a question of accessibility, but also a question of knowledge and skill (Norris, 2002). Access to digital tools that are powerful, without requiring technical skill, are needed to close this gap. This project concerns the creation of such a tool, which is easy to use for anyone while still being powerful and extensible.

1.3 Purpose

This report aims to describe the creation of a development platform, known as *Marker*, for data visualisation based on a plain text syntax. The formatting syntax should be easy to learn, understand and use. The platform shall be constructed in a way that allows other programmers to add additional methods of data visualisation. This platform will be implemented as a web application for inputting data and to create visualisations by wrapping the data in a syntax.

1.4 Scope

The project scope is limited to creation of a development platform by designing a text syntax, and to program a parser that generates JSON formatted data from that syntax. Some less complex methods of visualising data will be implemented in the context of the platform, which use the JSON formatted data. The platform will also be implemented in a JavaScript web application, that demonstrate some platform capabilities by wrapping a text editor and generated visualisations in a simple GUI.

2 Theory

The *Marker* platform contains several components that are built upon theory from different areas in computer science. This includes parsing, regular expressions and user interface design. These are all very different in nature, but they all require explanation in order to understand how the *Marker* platform is built.

2.1 Parsing expression grammar

A *parsing expression grammar*, hereinafter called PEG, is a formal grammar consisting of rules that define strings. A rule in this context, describes the structure of a string. PEG is often contrasted to *Context Free Grammar*, hereinafter called CFG. The main difference between these two is that PEG is not ambiguous, meaning that there is only one resulting tree structure. If PEG succeeds parsing the syntax, the result will be the first matching expression and the other results are ignored. PEG grammar is therefore strict while CFG offers multiple solutions. PEG is used to implement *Marker's* own parser upon which the application is built.

One major difference between PEG and Regular expressions, which will be described in detail in the next section, is that PEG does not skip characters in order to find expressions (Sigaud, 2015). PEG extends regular expressions and is therefore more powerful. PEG supports "ordered choices", which allows the parser to search for different rules in the same expression. This could be used when providing a data set that consist of both strings and integers. The expression `integer/double/string` will search for one of the predefined matching rules in the input. The number occurrences of each rule can also be specified in combination with the ordered choices (Sigaud, 2015).

2.2 Regular expressions

A regular expression describes a pattern of characters. Regular expressions can be used for parsing and identifying characters in strings (w3schools, 2015). Regular expressions are powerful and are of great value when parsing, due to their extensibility and modularity. Using regular expressions within rules in PEG extends the functionality of PEG.

Table 1: Regular expressions explanation table

Characters	[]
At least one	+
Result or null	?
Literal	"literal"
Zero or more	*

An example of a regular expression that checks for at least one character could therefore look like seen in listing 1:

```
[A-Za-z]+
```

Listing 1: Simple regular expression rule

The "+" character could be replaced with expressions like e.g. "?" or "*" from table 1. The strength of regular expressions appears when analyzing different data types as strings, doubles and integers etc.

JavaScript functions like replace, match and split are all based on regular expressions. Regular expressions could be objects in JavaScript. An object of this type can be created with a constructor as seen in listing 2 (Mozilla, 2015).

```
// JavaScript code
var RegExpObj = new RegExp("ab+c");
```

Listing 2: Example of object creation

The created object can be used to test strings. The method exec() will return the first match of an input string. The test() method will return a boolean if the string matches the regular expression (w3schools, 2015).

The function in listing 3 shows how the string "Excel" is replaced with the string "Marker".

```
//JavaScript code
var textStr = "Excel is great!";
var result = textStr.replace("Excel", "Marker");
```

Listing 3: Example of JavaScript string replacement

2.3 User interface design and usability

There are several important parts to consider when designing a user interface in order to maintain a high degree of usability. Usability describes the ease of use of some object and can in turn be divided into sub categories; effectiveness, efficiency, safety, utility, learnability and memorability. Theses categories must be taken into consideration when designing a user interface (Rogers, Sharp, and Preece, 2011).

- **Effectiveness** describes how well the system performs the tasks that it is supposed to support. Does the GUI allow users to perform the tasks? A poorly designed interface may interfere with function while a well designed one can enhance it.
- **Efficiency** is a way of quantifying the amount of resources a user has to consume in order to complete a task. Common tasks should be easy to complete and demand little resources. By resources, things such as time or similar are meant. In the case

of this project it could mean that a user creating a simple graph should be able to do so with only minimal effort. An example would be to use good default values for the graphics and thereby only requiring minimal input from the user in the form of data values. Entering only data into bar chart should still yield a good visual result.

- **Safety** alludes to avoiding undesirable situations for the user. A typical example is the use of a dialog box for certain actions, such as the confirmation dialog box when deleting a file from the hard drive of a computer. It is important that the user does not accidentally remove all work done and if it were to happen, that the user can recover as much as possible.
- **Utility** refers to at what extent the application offers the necessary features. A graph tool where the user cannot create a bar chart, which is a standard way of expressing data, lacks utility. This does not mean that the amount of features should be as high as possible. Superfluous features might cause confusion and distract the user from the more important parts.
- **Learnability** describes how easy the application is to learn. This is a crucial part of designing a GUI since a hard to learn interface quickly will turn off users. An easy to learn interface is often a familiar one, this means that convention should be followed where possible.
- **Memorability** is closely related to learnability and refers to how easy the application is to use again once it has been learned. The user should not have to re-learn the application every time it is used.

The colors used in a GUI are also important since different colors might give different conception and should not only be decided on because they look good. For example, a dark page can feel edgier, more somber or more energetic depending on other design aspects (Tidwell, 2010). A light page, on the other hand, gives a more spacious feeling. Certain colors also give different impressions and feelings. Despite being quite individual there are some general factors that can be used. For example, the color yellow is often related to warmth and cheerfulness (Cherry, 2015).

3 Method

The project was divided into a pre-study and the implementation of the system. The pre-study phase was used to gather both theoretical information about underlying topics as well as provide insight into the current technologies such as programming frameworks that might be of use to the project.

The pre-study was started of by investigating what the current tools in data visualisation are, what they are good at and more importantly what they are lacking in the context of our purpose. This was done primarily through qualitative analysis in the form of interviews with experts in the area and thereafter exploring this domain ourselves.

Another important aspect is how different types of data are structured and accessed. Since the application should be able to create more than just common bar charts, broader insight into data visualisation was needed. Thus, a hackathon was attended where large volumes of data from the Swedish authorities was made accessible and put to use by the attending programmers.

Lastly, a literature study was carried out in order to gain more in-depth knowledge about parsing, syntax construction and other relevant areas.

3.1 Interviews with data journalists

In order to assess the current situation of data visualisation tools we decided to contact a specific group of people; journalists working with data and data visualisation. So called data journalists often create advanced data visualisations in their work but are journalists rather than programmers and therefore part of our target group. They possess knowledge about the possibilities as well as drawbacks of the current tools. To learn more, we conducted two semi-structured interviews with knowledgeable people from this group. The interviewees were two journalists at the forefront of data driven journalism in Sweden; Kristoffer Sjöholm from *SVT Pejl* and Jens Finnäs from *Journalism++*. These interviews served the purpose of answering the following questions:

- What methods and tools are used today in the making of statistical graphics and visual representations?
- How well do the current tools work and what are the biggest issues?
- Is this project and it's philosophy relevant and viable?
- What type of data is used and from where is it obtained?
- Can a text based application be adopted by a community of non programmers?

According to Sjöholm and Finnäs, some of the commonly used tools today are *Highcharts*, *Data wrapper* and *Tableau.* Highcharts, while being used by many big companies, does not contain the simplicity we want since our intended user will not be programmers, but journalists and such. Looking at the syntax below in listing 4, one can see that creating a simple bar chart as in figure 1, requires specific prerequisites in programming. Even though one can see what the different parts of the syntax relates to, the structure is quite complex and the syntax itself is not forgiving, which means that small errors in the code, such as missing a colon, leads to the code not executing.

```
$(function () {
    $('#container').highcharts({
        chart: {
            type: 'column'
        },
        title: {
            text: 'Monthly Average Rainfall'
        },
        subtitle: {
            text: 'Source: WorldClimate.com'
        },
        xAxis: {
            categories: [
                'Jan'
            ],
            crosshair: true
        },
        yAxis: {
            min: 0,
            title: {
                text: 'Rainfall (mm)'
            }
        },
        plotOptions: {
            column: {
                pointPadding: 0.2,
                borderWidth: 0
            }
        },
        series: [{
            name: 'Tokyo',
            data: [49]
        }, {
            name: 'New York',
            data: [84]
        }, {
            name: 'London',
            data: [50]
        }, {
            name: 'Berlin',
            data: [43]
        }]
    });
});
```

Listing 4: Example of JavaScript bar chart creation function

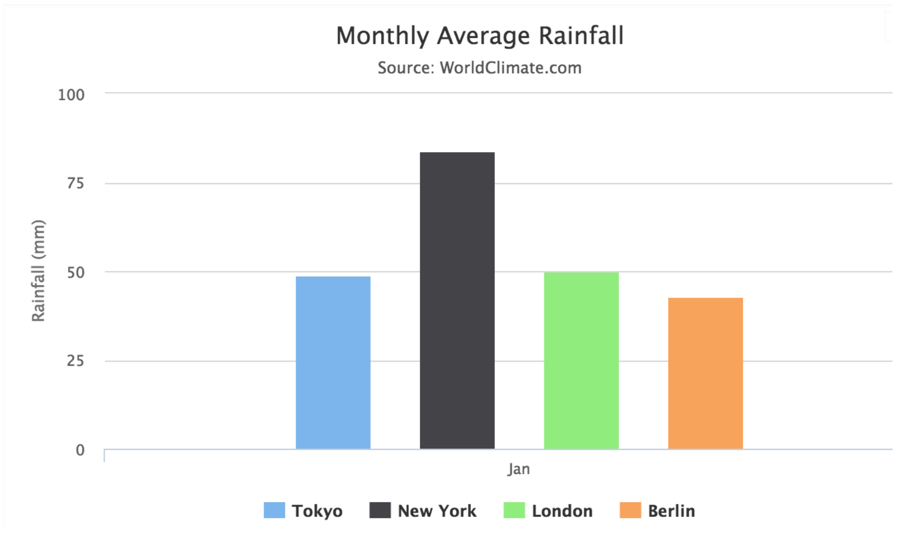


Figure 1: A simple bar chart created with Highcharts.

Similarly, the other programs mentioned in the interview have disadvantages as well. *Datawrapper* is quite simple, using a step-by-step wizard and checkboxes, radio buttons, small text input areas and drop down menus. Nevertheless, there is no way to add your own graphs and it does not support any type of customisation or in-depth features. *Tableau* is, in contrary to *Datawrapper*, a quite complex software aimed at businesses and offers more features than wanted in this project.

Summarizing existing tools, there exists many programs today that visualise data. However, most of these programs are directed at a different target group than intended for this project. Many of these are complex in nature and require knowledge about programming. The ones that are easy to use on the other hand, lack customisation and power.

The type of data that is used in visualisations today of course varies. But one standard way to store and represent data for a common chart such as a bar chart is comma-separated values. This is often saved as `.csv`-files. Comma-separated values can also be applied to other graphs, such as line charts, pie charts and more. Thus, this is a key part in representing data and adding support for these files is important for the market value of a data visualisation application.

One application that uses `.csv`-files is Excel. According to the interviewees, a lot of data is often stored in Excel or *Google Sheets*. This means that in order to address a lot of users, adding data from external sources such as Excel and *Google Sheets* should be supported. Linking a *Google Sheet* could also mean that data could be updated live. Data that changes over time is rarely used though, meaning that the data sources for the most part are static.

The interviewees also say that a text based application could be adopted by a community of non-programmers if made easy enough. The data journalists that were interviewed say that they are missing a service that creates a foundation quickly through templates but also is easy to customize later on. On a side note, they also mention that current tools lack support for annotations and highlighting, i.e. graphics on top of the visualisation such as arrows and high quality tooltips. It should also be possible to export the graphics as an iframe. An iframe is an HTML Inline Frame Element that represents a nested browsing context, effectively embedding another HTML page into the current page. Other than that, exporting graphics to host on your own server should also be supported. Existing tools also lack responsive design when exporting from current services.

Based on what Sjöholm and Finnäs have said, it is a fact that there is a need for a platform where non-programmers can translate their data into knowledge. This means that there exist a possible business case from the data visualisation tool being created. This will be further discussed later in the report but not taken into account during the time of the actual project.

Sjöholm and Finnäs both agreed on one thing as a conclusion; Generally, journalists know **nothing** about programming or similar. This information has to be taken into account during this project. From constructing a syntax to designing the GUI, simplicity needs to be prioritized.

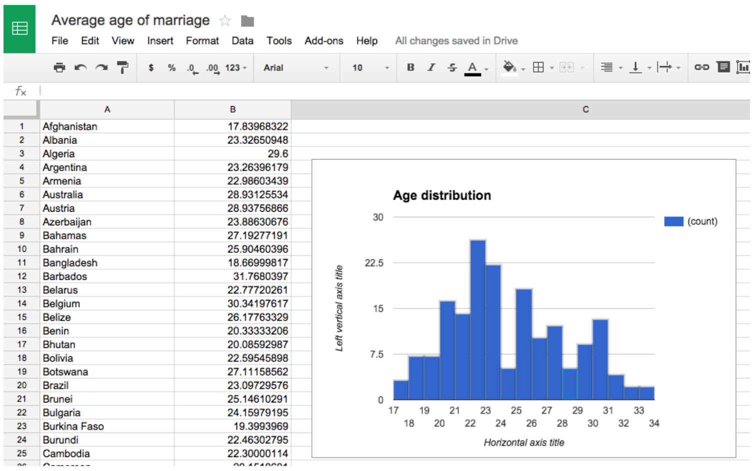
3.2 Analysis of data from Swedish administrative authorities

As part of our pre-study we also participated in a programming event hosted by the Swedish authorities in Stockholm. Among these authorities were *Lantmäteriet*, *SMHI*, *SCB* and *Skatteverket*. The authorities opened up large sets of data of various kinds with hopes that the participating programmers would make constructive use of it. This was a way for the team to assess what kind of data that can be used in visualisations and what the most common types are. It also served the purpose of a practice round in coding data driven applications. The event, considering the hosts, also indicates that the topic of this project is currently very relevant and not only of great interest to ourselves.

3.3 Analysis of popular data visualisation tools

One of the goals of the pre-study was to find out in which way the imagined tool could bring the most new value into the area of data usage and visualisation, thus our pre-study include an analysis of the most popular current tools, to be able to assess what is missing from them, and what could be improved upon.

- **Excel, Numbers and Google Sheets.** Excel, Google Sheets and Numbers are all popular programs with both data management and graphing capabilities. They import data both with proprietary formats, and plain text data like `.csv`-files. In some cases, like with *Google Sheets*, the data can be pulled from online services, such as online forms. Graphing capabilities include making the most common visualisations such as bar and pie-charts, and they are configured with a user interface consisting of checkboxes, dropdowns and text fields. An example of this can be seen in figure 2.



	A	B
1	Afghanistan	17.83968322
2	Albania	23.32650948
3	Algeria	29.6
4	Argentina	23.26396179
5	Armenia	22.98603439
6	Australia	28.93125534
7	Austria	28.93756866
8	Azerbaijan	23.88630676
9	Bahamas	27.19277191
10	Bahrain	25.90460396
11	Bangladesh	18.66999817
12	Barbados	31.7680397
13	Belarus	22.77720261
14	Belgium	30.34197617
15	Belize	26.17763329
16	Benin	20.33333206
17	Bhutan	20.08592987
18	Bolivia	22.59545898
19	Botswana	27.11158562
20	Brazil	23.09729576
21	Brunei	25.14610291
22	Bulgaria	24.15979195
23	Burkina Faso	19.3993969
24	Burundi	22.46302795
25	Cambodia	22.30000114

Figure 2: An Example spreadsheet with a graph, using Google Sheets

- **R Statistics.** In the statistical research community, R statistics is a popular programming language that puts emphasis on easy handling of data, and a widespread array of packages that can visualise data (“What is R?,” 2015) .
- **Processing.** Processing is a programming language aimed towards generating visuals. It is popular among professionals who want an advanced tool for visualisation, with interaction and animation possibilities (“Overview. A short introduction to the Processing software and projects from the community,” 2015).
- **Infogram.** Infogram is an online tool for generating small infographics quickly (“Infogram is the data visualization product that brings out the best in your data,” 2015).
- **Charts.js.** Charts.js is a JavaScript graphing framework, aimed towards web developers who want to have live graphs and charts on their webpages. Possibilities when using JavaScript frameworks include live generation of charts using arbitrary data available to the programmer building the website (“Simple, clean and engaging charts for designers and developers,” 2015).

4 System design and implementation

The system design of *Marker* is constructed to allow extensibility and flexibility by providing the possibility for third party developers to contribute additional graph modules. This is accomplished by separating the different components of the application and thereby allowing developers to connect their own components. When a developer creates such a graph-module, the developer has to register it to the *Marker* framework which then incorporates it into the system. Each graph-module also declares a small API (Application Program Interface) specific to that graph-module, stating what *selectors* and *properties* are customizable.

The design of the system can be abstracted as 5 different sub systems as illustrated in figure 3.

- **Syntax.** A syntax that is deemed easy and forgiving, with both flexibility as a development tool, and usability for users that might be non-programmers.
- **Parser.** A program that analyzes our syntax and converts it into a predictable and usable data structure.
- **JSON.** The output from the parser in the form of JSON. An intermediate form that links the syntax to the subsystem responsible for graphics.
- **Processing.** Program packages, made by us or others, that construct the desired data visualisation from the JSON data.
- **Visuals.** The final stage, the actual graphics generated from the processing step, that could be exported in various forms - either as static images or interactive web components.

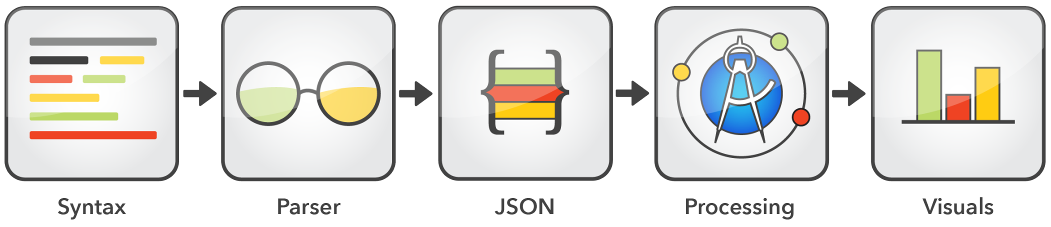


Figure 3: The main building blocks of the system.

4.1 Syntax design

When designing the syntax for the *Marker* system, the syntax was made as easy and usable as possible for new users. One way of doing this is to build on a foundation of what our users already know.

To accomplish this, platforms where non-programmers already have used a structured syntax were studied. Two example of this are *Markdown* and *LaTeX*, but another example are users of the social network *Twitter*, who use a syntax to format their *tweets*. This syntax looks as follows.

- `@`-symbols are used to target names of users of the Twitter platform.
- `#`-symbols are used to categorize the tweets with different keywords.

The at-sign is noted to act as a selection operator, as every at-name on twitter is a unique identifier. The number sign is noted to act as a categorizing operator, useful for searching for tweets on a specific topic.

This train of thought inspired the syntax design process, along with trying to make it feel less like a programming language and more like intuitively structured data about the visualisation. Care was also take to ensure that the syntax actually could be parsed into relevant structured data.

4.2 Generating a parser with PEG.js

PEG.js is a parser generator which uses *language grammar*, rules that dictate the semantics and syntax, for the JavaScript programming language. Given a grammar consisting of rules, the parser will return an output specified by the programmer ("Parser Generator for JavaScript," 2015).

PEG.js is easy to use and to install. PEG.js grammar can be compiled through PEG.js online tool ("Parser Generator for JavaScript," 2015) or compiled on the command line. The command in listing 5 requires an installation of PEG.js, which could be done simply with a package manager.

```
> pegjs grammar.pegjs
```

Listing 5: Compilation of the PEG.js grammar

Running this command from the terminal produces a JavaScript file containing the parser. If the grammar contains non valid syntax, the errors will appear in the command line window. The parser is simply a JavaScript function that takes a string as input, and outputs the result from the matched expressions.

Listing 6 shows one snippet of example grammar:

```
start = ((val:int "\n"*) { return val })+

int = digits:[0-9]+ {
  return {
    type: "int",
    value: parseInt(digits.join(""), 10)
  }
}
```

Listing 6: Example grammar snippet

Rules in PEG.js uses regular expressions to match expressions. For the parser to be able to look for expressions, a start rule must be defined. This rule defines the search pattern for the parser. The `int` rule from the example above matches input containing 0-9, labeled as digits. The plus sign inherits from regular expressions and represents "at least one", which means that at least one occurrence should be present in the input. The return statement is used for specifying which output should be presented. For each occurrence, the parser produces JSON-data in shape of an abstract syntax tree (AST), with the labels type and value. JavaScript code and functions can be used to modify the input which extends the functionality of the parser.

PEG.js is a powerful parser generator and is also easy to learn since it is mostly based on regular expressions and rules following the same structure. PEG.js offers intuitive and natural ways to specify a grammar. Rules, labels, expressions and operators are essential when defining a grammar with PEG.js.

- **Rules** A rule is unique and is followed by an equality sign. The `int` rule from the above example parses integers. Rules can refer to other rules. Rules in PEG.js can be recursive and can therefore be useful when searching for an arbitrary amount of data in a row, like arrays. Recursion and the ability to put matched expressions into variables leads to cleaner and a more intuitive grammar that is easy to follow.
- **Start rule** A start rule is obligatory and the first to be interpreted. Often referred to as `start`, as in the example above.
- **Labels** Labels are used for referencing matched expressions. In the `int` rule from the example above, the expression will be referenced as `digits`, which can be used in the return statement and could also be sent to an external JavaScript function.
- **Expressions** Several expressions can be used to determine number of occurrences of matched syntax. These expressions inherit from regular expressions. The expressions mentioned in table 1 in chapter 2.2 are equivalent.

A general rule could therefore look as in listing 7:

```
ruleName = reference:("expression") {return reference}
```

Listing 7: A general PEG.js expression rule

The parser could also be implemented directly in HTML or JavaScript file like in listing 8:

```
//JavaScript code
var parser = PEG.buildParser("rule = ('first' / 'second')*");
```

Listing 8: Parser in JavaScript code

The parser object could thereafter be used together with the function `parse` in order to parse the input.

```
//JavaScript code
parser.parse("Parse this string")
```

Listing 9: String parsing in JavaScript

4.3 The JavaScript framework Meteor

All the graph-components of *Marker* are created as *plain* JavaScript files. However, to make the graph-components, a user interface, an editor and a server component work together; They need to be bundled into a web application. Web applications are generally dependent on a web application framework, which are in turn bundles of code easing the process of making something interactive running on the internet.

This project utilizes a web application framework called Meteor. To get a grasp of what Meteor is, and why it suits this project, we must first take a look at something called Node.js.

Node.js is one of the most popular and extensive web application platform frameworks as of 2015 and is written solely in JavaScript. It offers low level control and high levels of customisation of all components in the application, and the user is obligated to configure the program-flow of databases, back-end and front-end components, and explicitly state how they are going to communicate with each other. In other words, in Node.js the user is responsible for the application architecture ("About Node.js," 2015).

Meteor is an open-source web application framework written on top of Node.js. It is an *abstraction* of Node.js, which bundles database, server and client logic in one framework. When deployed for production, the Meteor application *builds* itself into a Node.js application. In Meteor, the developer is not forced to configure how all the components are connected, almost all set-up is done automatically and lets the developer focus on creating the product from the start, instead of the application architecture. This allows for rapid prototyping. Meteor also produces cross-platform code, which means that the web application can be converted into a native iPhone– or Android application with minimal effort ("The Meteor mission," 2015). While not in the scope of this project, this could prove valuable for future development.

Meteor is, at the time of writing (March 27, 2015), the 10th most starred repository on the largest open source development site *GitHub.com* ("GitHub," 2015).

Meteor also features what is called a *package manager*. A package manager is a tool used for organising and distributing third party frameworks, and allows for easy extension of an application. For example, if a user needs authentication functionality in an app, the user can simply add a finished package offering login functionality, instead of implementing one. Meteors package manager is called Atmosphere and is operated via *command line input*. More information about Atmosphere can be found at **http://atmospherejs.com**.

At the time of writing (March 31, 2015), there exist over 4500 packages in Atmosphere. Several packages was used in the project to simplify styling, layout and functionality of the application. Installing a package through Meteors package manager is done by command line input as seen in listing 10:

```
> meteor install package-name
```

Listing 10: Installation of a Meteor package

Once this is done, Meteor will install the package and, given that the package itself works, it can be used straight away. Basically, this means that getting a package up and running within the project is extremely efficient.

4.4 The JavaScript framework D3.js and graphics generation

The graph components in *Marker* are contained within standard JavaScript-files and the creation of the visuals may be performed in what ever way is preferred by the developer. One of the most common tools for doing this and what is used throughout this project is the framework D3.js. It supplies means to establish a connection between individual elements from a data set to corresponding graphical components of the web site ("Data-Driven Documents," 2015).

D3.js does not provide or create any graphics by itself, instead, the framework works as a link between data and graphics. The graphics are created by using *SVG* (Scalable vector graphics) in conjunction with D3.js.

4.5 Jade, Jeet and Stylus

Some of the packages used extensively in the project are Jade, Jeet and Stylus. These packages are all used in order to simplify the development of *Marker's* front-end and does not add any extra functionality. Stylus and Jeet both serve the purpose of creating a layout and design. Ultimately, they both produce CSS code as output. Jade serves a similar purpose but instead creates the main markup of the page and outputs HTML-code.

Jade is a template engine which is used in conjunction with Meteor. Jade produces HTML code, and supports dynamic code and re-usability ("Jade Language Reference," 2015). Meteor has a good default template mechanic but using Jade removes some of the tedious work required with HTML syntax. Jade has a minimalistic syntax based on indentation levels, which is in contrast to the more verbose (i.e. many symbols) HTML that it outputs. Jade also incorporates logic such as loops and conditions which is essential when creating a non static web page. A simple page created with Jade might look like listing 11:

```
doctype html
html(lang="en")
  head
    title= pageTitle
    script(type='text/javascript').
      if (foo) bar(1 + 5)
  body
    h1 Jade - node template engine
    #container.col
      if youAreUsingJade
        p You are amazing
      else
        p Get on it!
      p.
        Jade is simple.
```

Listing 11: Simple web page made with Jade

The above Jade-code will produce the following HTML as seen in listing 12:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Jade</title>
    <script type="text/javascript">
      if (foo) bar(1 + 5)
    </script>
  </head>
  <body>
    <h1>Jade - node template engine</h1>
    <div id="container" class="col">
      <p>You are amazing</p>
      <p>Jade is simple.</p>
    </div>
  </body>
</html>
```

Listing 12: Example of how Jade will be compiled into HTML

Stylus is a style sheet pre-processor meaning that it ultimately outputs CSS. The point of using such a pre-processor is that it makes the writing of complex CSS easy. Amongst other things, Stylus makes it possible to reuse code, define variables and functions and evaluate mathematical expressions ("Stylus - Expressive, dynamic, robust CSS," 2015). On top of this, Stylus uses a similar syntax as Jade in the sense that it is indentation based.

```
a
    color black
    font-size 12px

h1
    color blue
    font-size 28px
```

Listing 13: Example of Stylus syntax

The above Stylus-code in listing 13 will produce the following CSS in listing 14:

```
a{
    color: black;
    font-size: 12px;
}

h1{
    color: blue;
    font-size: 28px;
}
```

Listing 14: Simple CSS code

Jeet is a framework that is used in order to create layout columns and grids on a web page. By providing Jeet with either a decimal number or a fraction, it creates a column of corresponding width (“A grid system for humans,” 2015). In order to create a column that spans half of the page, the following expression simply has to be added to the style sheet file as seen in listing 15:

```
.a-class-in-stylus
    col(1/2)
```

Listing 15: Simple Jeet example

This expression will evaluate to more complex CSS code. Hence, Jeet does not extend CSS but makes it easier to both write and read. The effect of the code is that every HTML element with the CSS-class *a-class-in-stylus* now will span half of the web page.

4.6 Ace - Web code editor

Ace, is an open-source JavaScript framework for integrating a powerful code editor to a website. In the application, Ace will be used as a standard text editor where a user enters syntax in order to visualise their data. It supports most features that developers are used to, such as code indentation, syntax highlighting and search-and-replace. It also supports working with large documents of text without slowing down the web browser (“The high performance code editor for the web,” 2015).

5 Results

The result of this project is *Marker v0.1*, a web application residing on the domain ***markerapp.meteor.com***. The complete application includes several modules and vital parts which will all be explained in this section.

5.1 Marker syntax

The primary goal of constructing the syntax was to strive for both understanding and intuition for a non-programmer and maximum extensibility for users with coding experience.

The *Marker* syntax follows similar design patterns mentioned in 4.1 to make it more appealing and intuitive for the large audience. Its primary operators are *number sign* and *at-symbols*. The at-symbols are used for *selection* and the number sign is used for *formatting*.

For example, the at-sign could *select* an element that is exposed by the creator of a specific graph. E.g. `@highest` could select the bar with the highest value in a bar chart. The user of the language is then allowed to send (pre-defined) *CSS* properties to the specific element using indentation under the `@highest` keyword, perhaps to make its label bold or underlined.

The syntax is interpreted sequentially and designed to be very user friendly and allows the user to make small mistakes without any difference in the parsed output. The string `#data` must occur in order for the expression to succeed. The data field could be a comment, so the labels are passed to the function `commentOrValue` in order to determine this. The data set could be doubles, integers, characters, arrays of data and file names. The input could be a combination of these types, which the ordered choice operator `/` specifies. The formal language grammar uses regular expressions within the rules to distinguish between the syntax components.

With support for both *csv* (Comma-separated values) and *json* files, the user can directly pass a local stored file through the syntax to the graph instead of manually type all values. This can be done in the following way after the user has selected the corresponding file via the *Import Data* button. And the syntax could look like in listing 16

```
#data: weather_data.json
#title: My Chart
```

Listing 16: Simple *Marker* syntax of external data import

5.2 Marker parser

The parser is implemented using PEG.js and the grammar is a combination of JavaScript and regular expressions as mentioned in section 2.2. The parser input is interpreted and parsed towards the formal language grammar. Each input that matches a given rule produces an output. Each data type that is specified in the grammar is defined as a rule. A matched expression returns JSON-data, which structure is defined in the return statement in the rule. The intended use of the resulting JSON tree is as input data to the *Marker atoms*, which will visualise the data.

The data set is typed after the `#data` syntax. The `data` rule below in listing 17 validates the syntax towards matched expressions. The `string`, `file`, `double` and `int` rules will all return arrays which will form the AST, while `chars` rule just return regular characters.

```
data = comm:("--")* _ '#data' _ array:((" "* (nlTab/',')? nlTab? " "* (
    string / file / double / int / chars) (' '/'\t')* '\n'?)+) ';'? (nlTab
    *)? {
  var newArray = new Array([]);
  var valueArray = [];
  var tempArray = [];
  var separatorArray = [];
  var arrayCounter = 0;

  function createArray(array) {
    (array ? array.map(function(array) { valueArray.push(array[4]);
        separatorArray.push(array[6]) }) : null)
    for(var i = 0; i < valueArray.length; i++) {
      tempArray.push(valueArray[i]);
      if (separatorArray[i] != null) {
        newArray[arrayCounter] = tempArray;
        tempArray = [];
        arrayCounter++;
      }
    }

    if (newArray.length > 1) {
      return newArray;
    } else {
      return valueArray;
    }
  }
  return commentOrValue(comm, "data", createArray(array))
}
```

Listing 17: PEG.js grammar of the `#data` expression

The underscore (_) character rule, shown below in listing 18, is commonly used in the *Marker's* language grammar and it makes sure the parser does not crash, e.g. if the user types an extra space character or misses a comma character. The `nlTab` commando simply ignores new lines and tabs.

```
space = [(" ") / ("\t")]*
_ = (space ':'? '='? space) {
  return null
}
```

Listing 18: Grammar rule over how space and tab should be handled

Below is an example that shows how user friendly the parser actually is. Following inputs in listing 19, 20 and 21 will be generated to an identical abstract syntax tree:

The first example:

```
-- This is a comment
#data: A,1,B,2
#title: My Chart
```

Listing 19: First *Marker* example syntax

The second example:

```
-- This is another comment
#data =     A 1 B, 2;
#title      My Chart
```

Listing 20: Second *Marker* example syntax

The third example:

```
-- This is a third comment
#data
    A
    1
    B
    2

#title My Chart
```

Listing 21: Third *Marker* example syntax

Notice that it does not matter for the user if they type tokens like :/=/, etc. The regular expressions handles all these types of combinations and filter them out.

After the syntax has been parsed, the parser will output an abstract syntax tree in the form of a JSON value as seen in listing 23. It is this data structure that the graphics are based upon. With the following syntax input as in listing 22:

```
-- This is a comment
#data
    'Gothenburg, Sweden' 500
    'Stockholm, Sweden' 1000
    'Berlin, Germany' 3502

#title Population in big cities x1000
#type BarChart
```

Listing 22: *Marker* syntax over a simple bar chart

The following output will be generated:

```
[
   {
      "type": "comment",
      "value": null
   },
   {
      "type": "data",
      "value": [
         {
            "type": "string",
            "value": "Gothenburg, Sweden"
         },
         {
            "type": "int",
            "value": 500
         },
         {
            "type": "string",
            "value": "Stockholm, Sweden"
         },
         {
            "type": "int",
            "value": 1000
         },
         {
            "type": "string",
            "value": "Berlin, Germany"
         },
         {
            "type": "int",
            "value": 3502
         }
      ]
   },
   {
      "type": "title",
      "value": "Population in big cities x1000"
   },
   {
      "type": "type",
      "value": "BarChart"
   }
]
```

Listing 23: Abstract syntax tree in JSON format

After the JSON-structure has been created by the parser, the data have to be processed in JavaScript in order to draw the corresponding chart. For this purpose there is a processing algorithm which do exactly this. Please see Appendix C for the full code reference. At first, the loop will look for the data array and then loop through that array to collect all input data and store it in a new array. This array will then later be the input to the chart. Later, the first loop will look for the chart type to be drawn.

5.3 Marker graph modules

Integrating new graph modules is easy with *Marker*. A single graph module is referred to as an *atom* within the *Marker* API. These *atoms* can be developed by anyone but they need to register themselves to a central unit called the *Molecule*.

The *atom* needs to implement the methods below and follow a specific design pattern.

- **init** The init method is called by the *Marker* application when the atom is initiated. It takes JSON-data, options and a callback as parameters.
- **draw** The draw method is called by the *Marker* application when the visualisations need to be initially drawn or updated if the values in the syntax has changed. It takes a `paper` as a parameter. The *atom* is responsible for appending its visuals to that `paper`.

An object named `info` supplies a description of the graph to the user, it needs to contain a *name* and a description of the *atom*. It is important to note that the name is case sensitive, as it is used in the syntax for using calling a specific *atom*. Then follows a function, or a *closure*, named after the *atom* that is created. The closure wraps the whole functionality of the *atom* and will later be exported as an object and registered to the Molecule to become part of *Marker*. The closure contains two private variables: `data` and `defaults`. The variable `data` stores the data set to be visualised, it will be assigned in the `init` method.

The variable `defaults` is the key to user customisation from the syntax level. It is an object that provides all mutable style values of the graph such as colors, title and labels. This object is the declared *API* of the atom, i.e. the *properties* of the *atom* the creator of it has exposed for modification to the end user. All properties sent to *init* as options will be overwritten to the *defaults* object, the *key* in option has to have the same name as a *key* in *defaults* for changes to affect the atom. The scenario below shows how this looks in practise.

The defaults object in an arbitrary *atom*, before modification in listing 24.

```
ArbitraryAtom = function(){
    var defaults = {
        title: "Default title",
        titleColor: "red"
    };

    // (Rest of the atom code is omitted)
}
```

Listing 24: Default *atom* function

A property named *title* is specified in the *Marker* syntax as follows in listing 25:

```
#title Weather
```

Listing 25: Example of *Marker* `#title` attribute

The property *title* is then overridden in the *defaults* object.

```
ArbitraryAtom = function(){
    var defaults = {
        title: "Weather",
        titleColor: "red"
    };

    // (Rest of the atom code is omitted)
}
```

Listing 26: Updated *atom* defaults

The `draw` method is were the data is turned into graphics. Primarily this is done through the use of D3.js, but it does not matter which software is used, as long as the contents is appended to the `paper` parameter, which is an *SVG HTML-node.*

Last but not least, the `BarChart` closure is exported as an immutable object, its required methods made public, and registered to the *Molecule.* This is done in the *atom* file calling the `registerAtom` method on the global *Molecule* object, and sending the atom closure and info as parameters.

The complete implementation of a bar chart *atom* can be found in Appendix B.

5.4 Marker visuals

From the text editor the user can program charts and diagrams which are displayed directly in the browser. In comparison to the *city weather* example in section 3.1, figure 1, the *Marker* syntax for similar data could be written as in listing 27:

```
#data
    Tokyo   49
    NewYork 84
    London  50
    Berlin  43

#title Weather
#type BarChart
```

Listing 27: Example *Marker* syntax of a bar chart

Which renders the following chart as seen in figure 4.

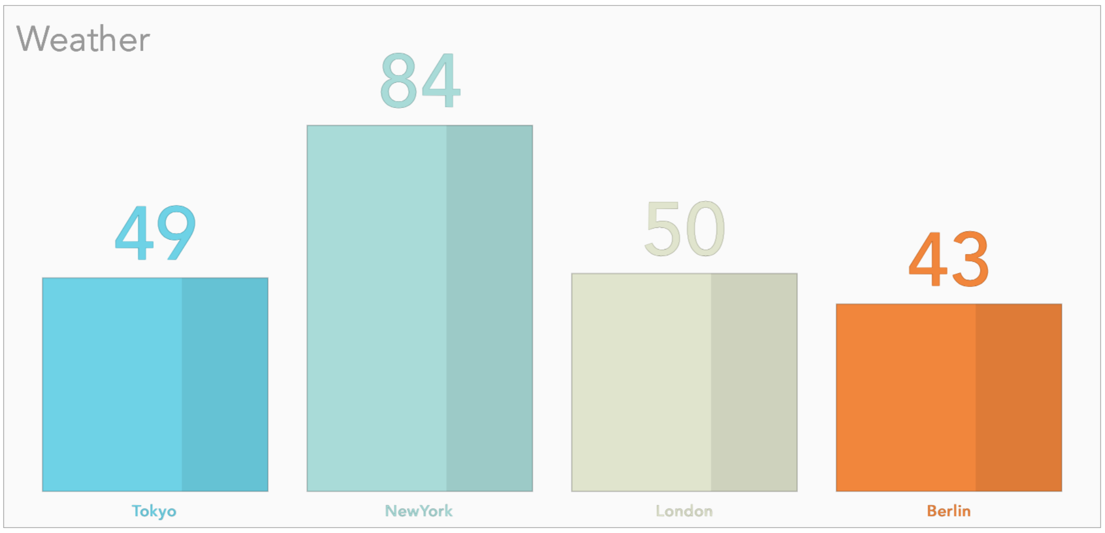
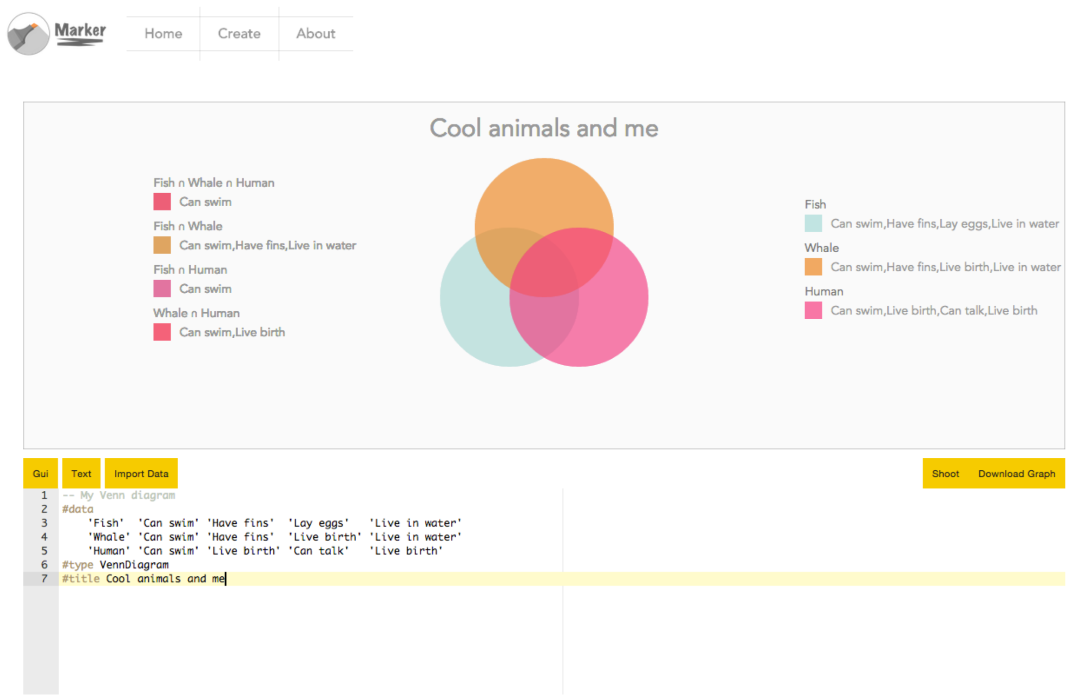


Figure 4: City weather graph from *Marker* syntax

5.5 Marker user interface

The user interface of *Marker* is based on a text editor and a view of the generated graphs as seen in figure 5. The editor supports color highlighting of the syntax according to our syntax specifications, as well as auto completion of properties. These two features increase typing efficiency and flattens the learning curve of the application. The highlighting provides visual feedback that makes the code easier to read, meaning that less resources has to be spent on this part. It also becomes evident if the user writes ill-formed syntax since this will not highlight. The auto completion partly has the same effect on efficiency since typing becomes quicker but it is also useful to new users. They can use the auto completion to get a list of possible alternatives when typing and thereby explore the full syntax.

When the user has written a segment of code and wants to view the resulting graph, he can do so by evaluating the code. This is done through the keyboard shortcut *cmd+return* for OS X users and *ctrl+return* for Windows users. Evaluation will instantly show the resulting visualisation over the text editor.



```
-- My Venn diagram
#data
    'Fish'  'Can swim' 'Have fins'  'Lay eggs'   'Live in water'
    'Whale' 'Can swim' 'Have fins'  'Live birth' 'Live in water'
    'Human' 'Can swim' 'Live birth' 'Can talk'   'Live birth'
#type VennDiagram
#title Cool animals and me
```

Figure 5: A view of the *Marker* GUI where a Venn diagram is being created.

6 Discussion

The purpose of this project was to create a development platform where a user without programming knowledge should be able to format and visualise data, using the simple plain text syntax developed for the project. The project included background research, with interviews and market analysis, which laid the groundwork for design of a useful platform for this purpose, that had the potential to bring value to the area of data visualisation.

After the design phase, the platform was implemented according to the proposed design. Being a platform, the implementation had to support many use cases, ranging from usability from the perspective of a programmer that want to build a custom visualisation, to a non-technical user that want to visualise their data. This wide scope was picked since early hypotheses concluded that this would make this type of application valuable - the wide variety of potential visualisations that are useful, make a niche application less interesting. This somewhat spread the project groups efforts thin, which affected to what extent the purposed could be fulfilled.

6.1 Assessment of outcome

The project resulted in a development platform for data visualisations, and a web application that implements this framework. The application is not feature complete at the time of writing, i.e. more development is required to make *Marker* full-fledged. However, it defines the core structure for both the syntax and the parser as well as the way in which graph modules may be integrated. In doing so, it also fulfills the major part of the thesis purpose. The different parts of the system were mostly developed in parallel in order to increase efficiency but this also became an issue when dependant parts of the system were out of sync. E.g developing a new graph module requires that the parser supports the data type to be used. This led to the fact that graph modules took longer time to develop than initially expected.

A significant part of the project's purpose was to create something that was user friendly, yet powerful. The applications performance in terms of usability can not be evaluated since user tests unfortunately are yet to be performed, thus, only the functionality itself may be evaluated. As of now, the application is developed in a way that the project group think our intended end users will be using the product. Having more user tests during the development and, especially, prototyping phase would have helped shape the project in a direction that could have further encouraged the end users to use our product.

Another significant part of the project was to support extensibility of the platform by other programmers, for adding more types of data visualisations, *atoms*. This is very much supported by the resulting platform technically, but several efforts remain for the platform to fulfill its potential here. Adding *atoms* require a programmer to do a *pull request* on *Github* with the new *atom*, that the project group then would accept into the platform. This process could be done in an more open manner, not requiring active participation from the project group. There is also potential for lessening the effort needed for making the actual *atom*, for example by improving the JSON value generated from the parser.

6.2 Implementation choices

One of the defining decisions in the project was to implement the application as a web application rather than a desktop application. This will hopefully increase spontaneous use of our tool since no installation is needed, which might help in acquiring new users. Since the data used in most cases will come from a web source and the publishing often takes place online as well, keeping the application within the same domain is a logical design choice.

Another important decision was to use Meteor as the main web application framework for the project. This has proved to be efficient since many components of a complete web application is already provided by Meteor. However, it was not without drawbacks, mainly concerning the compatibility between the Ace editor and Meteor, both very complex frameworks. There exists a number of different Ace editor packages for Meteor that all can be found using the official Meteor package manager, found on **http://www.atmospherejs.com**. Some of these were added to the project but later removed since Meteor syntax had to be used to interact with these packages. For example, adding the project groups own syntax highlighting mode was a big problem and the Ace editor was instead added manually to the project. In the end, using Meteor seems like the right choice since gains and time savings heavily outweighed the compatibility problems.

Using Jade, Stylus and Jeet was another choice the project group made. Jeet as a grid system has been working like intended and no further reflection has been made concerning this. There exist a lot of different ways to use grids but Jeet, being both minimalistic and intuitive, has been fully functional. Stylus and Jade are both whitespace sensitive languages and made both CSS and HTML coding efficient, despite some of the project members not having used this before. The learning curves for these are quite low and the payoff for using them is quite big time-wise.

During the implementation of the parser, a choice was made between using PEG or CFG. The main difference between these two is that PEG is not ambiguous, meaning that there is only one resulting tree structure that matches the expression. The lack of ambiguity was not the main reason PEG was preferred. CFG parsers often offers the ability to prioritize resulting trees. But this was excluded since that extra logic is not needed. PEG's behaviour is optimal in its functionality due to *Marker's* need of a consistent resulting JSON structure. *Atoms* would become too complex if they would take the JSON tree structure into consideration each time processing the input. The advantage of using a parser generator like PEG.js is the modularity it delivers. PEG.js is also well documented and many examples are provided. In PEG.js it is also possible to name input parameters arbitrarily, which makes it simple and intuitive to follow the code.

The possibility to pass internal PEG.js variables to pure JavaScript functions really extends the functionality of the parser. This is a valuable feature since it allows the parser to be integrated with other systems. The parsed input could for example be passed to a JavaScript function that sends HTTP POST requests to other more complex back-end systems.

PEG.js ordered choices functionality is used for enabling multiple data types in the same data set. This is essential for the use of e.g. key-value data sets. This allows programmers to build and combine data sets within the *atom*. Having an easy and forgiving syntax leads to less parsing grammar, in contrast to more advanced syntax, where indentation and brackets are used. More advanced parsing could be used if support for more complex relational data sets were to be implemented.

The error handling in PEG.js does not work that well now since the error exceptions are difficult to interpret for a non programmer. The error handling could be extended to match *Marker's* syntax.

```
-- My chart
#data
    Sweden 10 & 2
    Norway 12
#type BarChart
#title Countries
```

Listing 28: Example *Marker* syntax

The above non valid syntax in listing 28 throws the following exception in listing 29:

```
Uncaught SyntaxError: Expected " ", "#", "#data", "’", ",", "--", ".", "/
    ", ":", ";", "=", "@", "\n", "\t", [(" ") \/ ("\t")], [(\n\t)], [A-Za-
    z(_)?], [\-0-9], [\-A-Za-z0-9_\/\/ .?,\xE4\xC5\xE5\xD6\xF6\xC4], [\-A-
    Za-z0-9_\/\/.?\xE4\xC5\xE5\xD6\xF6\xC4] or end of input but "&" found.
```

Listing 29: Example of syntax exception

which is not optimal for inexperienced users.

6.3 Comparison with other data visualisation tools

Even without user testing and evaluation, there are conclusions to be made about what the *Marker* platform technically supports, that makes it stand out from other data visualisation solutions. Feature-wise, the application is not the most advanced. There exist other applications that handle larger data sets, more complex data and has a bigger supply of graphics. But what makes *Marker* really stand out is the open source expandability of the application along with its syntax. Adding new graphs is easily done by a programmer with basic proficiency in JavaScript, and something not offered by other tools. Thanks to the system design and the syntax, these graphs may be used by anyone, even individuals without programming experience.

Yet another feature that is important to take into consideration is how much code is actually needed to create, for example, a standard bar chart. If compared to Highcharts, *Marker's* code is way more efficient to type and considered more intuitive as a non-programmer.

Summarizing, *Marker* offers a simplicity that is not available anywhere else. This is at the cost of giving up some of the complexity of what the visualisations can represent. However, the combination of being easy to use but still customisable is something that does not exist in current market applications.

6.4 Platform usefulness

Is there a need for this kind of platform that standardises a workflow for working with data, for the purpose of visualisation? On the web today, there are powerful frameworks available for programmers that let them work with data in applications, and effectively draw graphics.

D3.js is a popular example of this, where data binding to the web document view is standardized and simplified, and JSON is very much the *lingua franca* for working with data in web applications. However, these tools are only available to programmers. Programmers are also used to having access to a wide variety of open source code - program components that once built, is free to use for anyone, anywhere.

Building a platform that enables anyone to utilize these powerful methods, without previous programming knowledge has potential to be valuable, but what potentially is even more powerful is introducing a standard for open source pre-built visualisations - to enable the open source workflow of programmers for data visualisation purposes. The platform, designed and implemented in the project has the theoretical potential for this - but further development, testing and research is needed to draw precise conclusions about what exactly is the best design of such a platform.

Using a syntax as the basis for user input to our platform has drawbacks, such as the fact that the parser implementation has to support all possible data combinations that different *atoms* might want to utilize for structuring the data input. *Atoms* might want custom data combinations, and this has proven to be inflexible with the current implementation of the parser. This is a trade off that in a way simplifies the user interaction - the user do not have to select which columns of data that correspond to specific visualisation features, such as in *Microsoft Excel* or *Google Sheets.*

6.5 User interface design

One of the main components of the user interface is the text editor. There are different ways and tools for implementing such a text editor on a web page. Using the framework Ace is one of the most well known methods and one that is well documented which is the reason it was chosen for this project. While technically powerful, it might not be the easiest tool for a non-programmer to understand. On the other hand, a large part of the project is focused on the syntax developed specifically for this application.

Using a text syntax and presenting an empty text editor to the user may not be the optimal way of attracting users without programming experience. This could initially be daunting for the users. Different ways to solve this issue are discussed in the following section.

6.6 Future work

One of the most pressing shortcomings of the current state of the application is the lack of help for new users. The blank canvas of an empty plain text document is possibly very intimidating for beginners. This is in contrast to one of the main purposes to the project, that the application should be easy to use for new users and non-programmers.

Implementation of an "onboarding" experience, where the user is guided in a step-by-step fashion, to the features of the application, would help many. Similarly, a GUI on top of the plain text document, is also something that would add immense value to the application for these user categories. This would allow users to grow with the application and use more in depth features once they have learned more. Finally, one idea would be to work with auto completion in combination with a lot of documentation.

For a platform to be relevant, and used by other programmers, it is common to encourage the building of a community with tools such as wikis, forums and up-to-date documentation. Active participation in further development of the open source software is also crucial for keeping software useful, especially on the web where technology moves fast. With our current implementation, updates to the core platform is needed to add new *atoms*, so as of now updates is also required for integrating new visualisation methods.

Most of the design and functionality implemented in the platform is based on interviews with people working with data journalism that have code experience. These people are one part of the proposed userbase; the part which could successfully implement their own *atoms*. There are also the other half of the users which have no coding experience. Performing thorough usability testing on this part of the users would be needed on further development of the product. This is to make sure the application can appeal to people without coding experience.

Another consideration for further development is to build APIs for using the platform tools (syntax, atoms) with other programming languages. For example, languages such as C, R and Python are all very popular in the scientific community that uses data for research, and having the option to interface with those would possibly help winning over those users to the *Marker* platform.

6.7 Potential business models

When designing the *Marker* platform, many considerations were made about what would make the product bring new value to the market of data visualisation software, based on what was learned in the interviews. As a consequence of this, a successful project might yield a product that has market value, with potential to build a business upon.

The *Marker* platform is open source. This is very core to what makes the product valuable for different users, as described earlier, but it also makes it hard to sell the software as the actual code, since it is available for anyone to download and use for themselves, free of charge. So if the project group would like to build a business upon the *Marker* platform, it would have to look for other options than to simply charge for the software.

One option is to offer more advanced features in a premium version offering. This would be a closed-source program built on top of *Marker*, that adds more value to the software, while still building upon the open source core, and using all the advantages from that. This could also be extended to offering the software as a service, with premium features that include customer support, a cloud storage/rendering solution, closed APIs and more.

Another option is to utilize the fact that the project group are experts on using the software platform, and use it for consulting purposes. From the interviews with professionals in the business, we learned that there is a large market for consultants that help other businesses with data visualisation. By using our software and expertise of the platform, this would be a potential business.

The project group also discussed a third option, which is to build a marketplace (*"App store"*) for *atoms*, similar to how, for example, themes are sold for the WordPress blogging platform - but for data visualisation purposes. Here individuals wanting a very specific visualisation may purchase an *atom* which suits those purposes for a small sum - and the creators of the app store takes a cut.

7 Conclusion

In this report, implementation of a data visualisation development platform known as *Marker* has been discussed. An easy plain text syntax was developed in order to make the visualisation tool accessible for non-programmers. The syntax offers extended customisation for the graphs depending on the type of visual that is used. Parsing of the syntax and a base of standard graphs were all implemented and the complete platform was then wrapped as a web application currently hosted on **markerapp.meteor.com**. The platform allows third party developers to create and easily integrate additional graph modules.

The goals of the project were met but further development that adds more features and enhances the user experience would increase the value of the product. Areas such as the parser, documentation, user testing and support for external data sets could also be improved upon in order to increase the value of the platform.

With the simplicity of the syntax in combination with the expandable system design, *Marker* has the potential to become a valuable tool for data visualisation in a world where data is becoming increasingly relevant.

Bibliography

Cherry, K. (2015). Color psychology, how colors impact moods, feelings, and behaviors. Retrieved April 15, 2015, from **http://psychology.about.com/od/sensationandperception/a/color_yellow.htm**

Felix, S. (2012). How facebook is tracking your internet activity. *Business Insider*. Retrieved April 18, 2015, from **http://www.businessinsider.com/this-is-how-facebook-is-tracking-your-internet-activity-2012-9?IR=T**

Findahl, O. (2014). *Svenskarana och internet 2014*. Stiftelsen för internetinfrastruktur. Retrieved April 18, 2015, from **http://www.soi2014.se/sammanfattning/**

Mozilla. (2015). Regular expressions. Online tutorial. Retrieved May 12, 2015, from **https://developer.mozilla.org/en/docs/Web/JavaScript/Guide/Regular_Expressions**

Norris, P. (2002). *Digital divide, civic engagement, information poverty, and the internet worldwide* (1st ed.). Cambridge University Press. Retrieved April 23, 2015, from **http://www.hks.harvard.edu/fs/pnorris/Books/Digital%20Divide.htm**

A grid system for humans. (2015). Home page of software. Retrieved April 20, 2015, from **www.jeet.gs**

About Node.js. (2015). Home page of software. Retrieved May 8, 2015, from **www.nodejs.org/about**

Data-Driven Documents. (2015). Home page of software. Retrieved May 10, 2015, from **http://d3js.org/**

European Union Open Data Portal. (2015). Home page of organisation. Retrieved May 18, 2015, from **http://open-data.europa.eu/en/about**

Getting the gist of Markdown's formatting syntax. (2015). Home page of software. Retrieved May 10, 2015, from **http://daringfireball.net/projects/markdown/basics**

GitHub. (2015). Online search. Retrieved May 10, 2015, from **https://github.com/search?q=stars:%3E1&s=stars&type=Repositories**

Infogram is the data visualization product that brings out the best in your data. (2015). Home page of software. Retrieved May 7, 2015, from **www.infogr.am/about-us**

Jade Language Reference. (2015). Home page of software. Retrieved April 20, 2015, from **http://jade-lang.com/reference/**

Open Knowledge. (2015). Home page of organisation. Retrieved May 18, 2015, from **https://okfn.org/**

Overview. A short introduction to the Processing software and projects from the community. (2015). Home page of software. Retrieved May 7, 2015, from **https://processing.org/overview/**

Parser Generator for JavaScript. (2015). Home page of software. Retrieved May 7, 2015, from **www.pegjs.org**

Simple, clean and engaging charts for designers and developers. (2015). Home page of software. Retrieved May 7, 2015, from **www.chartjs.org**

Stylus - Expressive, dynamic, robust CSS. (2015). Home page of software. Retrieved April 20, 2015, from **https://learnboost.github.io/stylus/**

The high performance code editor for the web. (2015). Home page of software. Retrieved April 20, 2015, from **http://ace.c9.io/#nav=about**

The Meteor mission. (2015). Home page of software. Retrieved May 8, 2015, from **www.meteor.com/about**

What is R? (2015). Home page of software. Retrieved May 8, 2015, from **http://www.r-project.org/about.html**

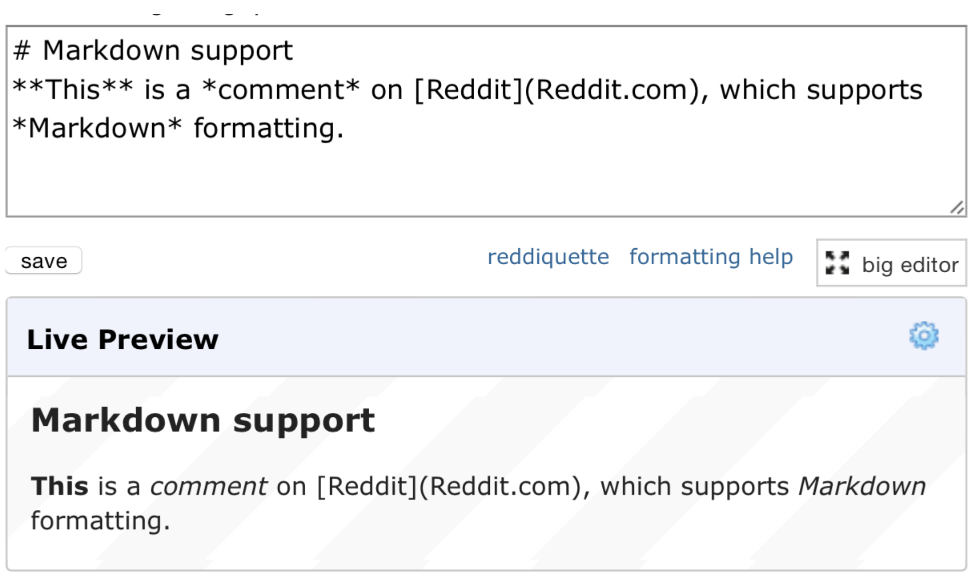
Rogers, Y., Sharp, H., & Preece, J. (2011). *Interaction design, beyond human computer interaction* (3rd ed.). Wiley.

Sigaud, P. (2015). Parsing expressions grammars basics. Online tutorial. Retrieved May 12, 2015, from **https://github.com/PhilippeSigaud/Pegged/wiki/PEG-Basics**

Tidwell, J. (2010). *Designing interfaces* (2nd ed.). O'Reilly Media.

w3schools. (2015). Javascript regexp reference. Online tutorial. Retrieved May 2, 2015, from **http://www.w3schools.com/jsref/jsref_obj_regexp.asp**

Appendix A: Markdown usage



```
# Markdown support
**This** is a *comment* on [Reddit](Reddit.com), which supports
*Markdown* formatting.
```

Markdown support

This is a *comment* on [Reddit](Reddit.com), which supports *Markdown* formatting.

Figure 6: An example of markdown usage, on the popular online community Reddit.

Appendix B: Barchart *atom*

```
var info = {
  name: "BarChart",
  description: "A barchart by the team"
};

function BarChart(){
  var data;

  // default properties for the atom
  var defaults = {
    chart:{
      background: "white",
      foreground: "#333"
    },
    lines:{},
    highest:{
      color:"red"
    },
    barWidth: 50,
    barMargin: 20,
    labelSize: 20
  };

  // called by library
  // onRecieveJSON
  var init = function (json, options, callback){

    data = json;

    if(options){
      //recursively merge defaults with options
      $.extend(true, defaults, options);
    };

    if(callback){
      callback();
    };
  };

  var draw = function (paper) {
    var paperHeight = parseInt(paper.attr('height'));

    var bars = paper.selectAll('g')
      .data(data)

    bars.enter()
      .append('rect')
        .attr('width', defaults.barWidth)
        .attr('x', function (d,i) {
          return i * (defaults.barWidth + defaults.barMargin);
        })
        .attr('fill', '#dd7777')
        .attr('height',0)
        .attr('y', paperHeight - defaults.labelSize)
        .transition()
        .attr('height', function (d) {
          return d.value;
        })
        .attr('y', function (d) {
          return paperHeight - d.value - defaults.labelSize*2;
        });

    bars.enter()
      .append('text')
        .text(function (d) {
          return d.label;
        })
        .attr('x', function (d,i) {
          return i * (defaults.barWidth + defaults.barMargin);
        })
        .attr('y', function (d) {
          return paperHeight - defaults.labelSize;
        })
        .attr('fill','black')
  };

  // init:init
  return Object.freeze({
    init: init,
    draw: draw,
    getDefaults: function(){return defaults;}
  });
};

Molecule.registerAtom(BarChart, info);
```

Appendix C: Marker processing algorithm

```
// Get parsed text from editor and parse it with peg.js
var parsed = parser.parse(text)
var data = [];
var chartType = "";

for (var i = 0; i < parsed.length; i++) {
    if (parsed[i].type == "data") {
      if (parsed[i].value[0][0] && parsed[i].value[0][0] !== "undefined")
          {
        // 2D array, we need to go deeper into the tree structure
        var dataArray = parsed[i].value;

        // Loop over all elements/arrays in data array
        for (var j = 0; j < dataArray.length; j++) {
          var temp = [];

          // Insert every data value into temporary array
          for (var k = 0; k < dataArray[j].length; k++) {
            temp[k] = dataArray[j][k].value;
          }

            // Assume label in data[j][0]. Values in rest.
            if (temp.length > 2) {
              // Insert temp array into data array. Remove first element
                  and add the rest
            data.push({label: temp.shift(), value: temp});
          } else {
            // Do not create new array, just return the values from temp
                array
            data.push({label: temp[0], value: temp[1]});
          }
        }
      } else {
        // Just get the values straight out of AST tree and insert into
            data
            for (var j = 0; j < parsed[i].value.length; j++) {
                if (parsed[i].value[i].type == "file") {
                    data = "";
                    data = parsed[i].value[0].value;
                    break;
                } else if (parsed[i].value[i].type == ("int" || "string"
                   || "double")) {
                    data[j] = parsed[i].value[j].value;
                }
            }
        }
    } else if (parsed[i].type == "type") {
      // Chech what chart type we want to draw
        chartType = parsed[i].value;
    }
}
```

```
var options = {};

// Collect all options as title, color, label etc and insert to options
    array
for (var i = 0; i < parsed.length; i++) {
  options[parsed[i].type] = parsed[i].value;
};
delete options.data;
```