An Online tool supporting the creation of Syntax Diagrams

Colm Mathews
13440022

Final Year Project – **2017**
B.Sc. Double Honours in Computer Science/Mathematical Physics

Department of Computer Science

Maynooth University

Maynooth, Co. Kildare

Ireland

A thesis submitted in partial fulfilment of the requirements for the B.Sc. Double Honours in Computer Science/Mathematical Physics

Supervisor: **Dr. John G. Keating**

Table of Contents

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study as part of this B.Sc. Double Honours in Computer Science/Mathematical Physics qualification, is *entirely* my own work and has not been taken from the work of others - save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed: [illegible] Date: 17/03/2017

Abstract

Systemic Grammar is a branch of linguistics that focuses on exploring the function of language, not only through its content, but the manner of its execution. Systemic Functional Grammar Trees (SFG Trees) are the most effective way of visualising a breakdown of this analysis. Visualisation is one of the most effective pedagogical tools for students, making the trees very useful in this regard. This project aims to develop a web application allowing the user to upload a text, isolate and annotate it in a manner relevant to SFG, and use this data to generate and display these trees to the user. The documentation, design, development and evaluation of this application is contained in this report. This project delivers user friendly applications on two platforms allowing users to upload and tag text, download tagging data, and display syntax trees, as well as box and theme diagrams. Syntax trees produced bear close visual similarities to the inspirational examples, maintaining data integrity and composition. The web application is light, portable, and easily extensible, making it a sound base to be built upon for future development of Systemic Functional Analysis tools.

Chapter one: Introduction

1.1 Goal

This goal of this project is to create a robust web application allowing the creation of systemic functional grammar analysis trees. The tree diagrams are to be created from information taken from user input, in the form of text annotation, i.e. 'tagging' the text. A utility to generate box diagrams and theme analysis tables is also required, as this would assist with SFL (Systemic Functional Linguistic) analysis of the text.

List of Requirements

- Utility allowing user to annotate text
- Saving, editing and deletion of such annotations
- Generation of Syntax Tree structure from annotated text
- Display of these trees to the user

Desirable Features

- Lightweight and simplistic UI
- Generation of Theme Analysis Diagrams
- Generation of Box Diagrams
- Flexible implementation, allowing easy extension
- The option to save and load annotation data to a transparent format for simple transit between applications

1.2 Motivation

Systemic Functional Grammar (SFG) is a form of grammatical analysis pioneered by M.A.K Halliday in the early seventies as a way of exploring the function of language not only by its content, but the manner of its execution [1]. In order to fully understand the meaning of a text, it is necessary to identify its functional components; either ideational, interpersonal, or textual. As explained by David Butt; "we really only understand other speakers when we share, not only our words and grammar but also *which* words and *which* grammatical choices are appropriate for a situation." [2]

In an educational environment, SFG can play a positive role in school contexts, as described by Zhiwen Fang, that "With functional grammar, language teaching and learning will be more interesting and meaningful". [3]

While there is an abundance of SFL tools, there is a distinct lack of *pedagogical* tools to assist in analysis of this type, and none that provide the ability to construct SFG trees from a tagged text. Improved pedagogical tools will lead to more efficient and effective learning of these techniques, which are becoming increasingly relevant and useful. Approaching text in this manner not only makes the way we understand and learn language more transparent, but will be invaluable in the future as machines learn to identify the words we are saying, but struggle to understand what those words mean in the context they are given.

1.3 Problem statement

There are various problems to address in this project, of varying complexity. The first of which was developing, or identifying a javascript library which would allow me to tag text for an unspecified duration, and identify it by other text, specified by the type of analysis being done. The library in question thus had to preserve core data about the text being tagged, namely, the text, the tag and the relative position and length of the tag in question. Next, an appropriate tree generation library had to be found which could provide the functionality required. On completion of these two tasks, the tag data had to be sorted by position and range in the document and assembled into a nodal tree structure that could be used by the tree library to generate the correct SFG diagram. Likewise, there had to be appropriate manipulation of the tag data to give correct box diagrams and theme analysis tables.

Secondary to the above primary functions, a coherent, functional and attractive user interface had to be developed to make its use desirable by any prospective grammar analysts that would consider using it.

1.4 Approach

It was necessary to familiarise myself with the basics of Systemic Functional Linguistics, and the various libraries available for javascript allowing the addition of the features that were required, either by the base functionality or with minimal additional work by myself.

An agile software development methodology was applied to the project, which allowed the greatest flexibility and results. It's avocations of adaptive planning, evolutionary development and continuous improvement were well suited to this project due to the lack of work done in this field prior. As I tried different development routes, it permitted me to adapt my plan as I went, learning from failures and dead ends in my coding and reasoning. I was free in the choices I made when choosing which libraries to use, but took care to remain mindful of the end goals of the project.

Tasks were assigned an order of priority, from highest to lowest, and as many as possible were completed in each incremental development phase, each lasting 1-3 weeks.

Agile development is also suitable in that it does not need all requirements to be fixed before the start. This allowed the project to be started on functional requirements, and gave the flexibility to allow addition of extra functional and non-functional requirements thereafter.

As the front end of the application was mostly HTML, Javascript, and CSS (HTML/JS/CSS), I used Sublime Text as my main editor, it was useful thanks to its native autocompleting features and error highlighting. Mozilla Firefox was used to periodically check the results of changing or adding code, making this the recommended browser for the application.

An XAMPP installation was used on my own personal laptop to interact with the server for the saving and loading of tagging data. When I had implemented the XAMPP server, and had adequately interfaced the front end of the project with a MySQL database through PHP, I coded a RESTful API allowing efficient communication between both sides of the project. On completion of this task, I turned my attention to researching a back end utilizing Node.js, which would provide some choice for any prospective developers using the application in the future.

The Node.js installation was well suited to the project due to its non-blocking, and event oriented nature. I installed Node on my machine, and thanks to its flexibility, was relatively simple to start a parallel server on the adjacent port to XAMPP, even allowing me to use `/htdocs` (home directory of XAMPP) as the home directory of the Node installation.

Unit testing was integral in the testing of functionality as I continued to add new features, testing their satisfactory operation. Integration testing was equally important, ensuring the integrity of existing features and functionality. Periodically testing integration ascertained the correct function of the software, and was paramount in establishing system quality by the end.

1.5 Metrics

My diagrams generated by the end creation will be compared to their original counterparts in Lise Fontaine's work [4], and assessed on their similarity. I will provide side by side comparisons of the end result and the target diagrams in the evaluation section of the report. This assessment of the diagrams in terms of similarity to the specification will dictate the degree of success of the project

Hence, functional testing will be one of the best indications of how successful and/or useful the end product is. I do not readily have access to any functional linguists, however my supervisor has not only experience, but an avid interest in linguistics, and maintains occasional contact with Lise Fontaine, author of the core text inspiring this project [4]. I will use this experience to my advantage, and analyse my work with my own understanding of the process as clarified by Lise Fontaine [4].

1.6 Project

This project culminated in four main key deliverables;

- Full WAMP stack (realised through XAMPP) hosting the completed product, closely matching the original specification
- Full Js framework, through Node, Javascript and Json, also hosting the completed product; changed only in the destination of the post requests
- A variant of the application independent of any server framework, allowing upload and download of annotation data, usable offline with no internet connection
- Yet another variant built upon the software framework Electron, facilitating the production of desktop applications for Windows, MacOS, and Linux (based upon the offline variant)

The completion of this project provides the first application to allow manual generation of syntax trees, on any platform. Its existence can now cater for the needs of systemic functional linguists, allowing generation of syntax trees, box diagrams and theme diagrams, using a light and simple UI. This was possible through manipulating and maintaining the structural integrity of abstract data structures. SFLers can use this service to focus on the way language is put together, and analyse how the meaning is communicated for particular purposes.

Using my Javascript and API manipulation skills, I had to develop my own storage plugin for annotator.js, which I found awkward and unintuitive. By creating my own plugin for data storage, and by adjusting and tailoring within the different versions of the application, I can now easily explain the use of storage within annotator, as well as the use of the API in general. I intend to condense this information, as well as the lessons I learned using annotator, into an instructional format, and make it readily available to other budding developers that wish to develop new plugins for it.

Chapter two: Technical Background

2.1 Topic material

The most important piece of topical material at my disposal was a book by Lise Fontaine [4]. This book in many ways lay the groundwork for the theory that the project is based on, and gives many examples of the diagrams and methods of analysis that are desired. This book proved particularly useful in showing how tree diagrams could be generated from pre-identified text. These lessons allowed me to develop a technical competency in basic SFL, and provided more than enough understanding with which to approach, and fabricate a solution to, this problem.

It explains how to go about isolating and identifying the constituent parts of a text, first establishing the largest units, such as clauses, moving on to the pivotal element, the main verb. The analyst then systematically iterates through the smaller units, determining the presence of groups, and the actions of the words within those groups.

From these explanations and examples, I was able to observe that the route of analysis classified the elements from largest to smallest, and concluded that my manner of input had to account for this.

In my research, I could not find any software that could do, or even claim to do what I aspire to achieve in this project. My supervisor provided anecdotes of grammar conferences where the highest tech tool provided there were pieces of stationery and writing utensils.

The closest counterpart I could find online was a utility offered by fox type, allowing generation of a tree analysing typed text. This differs from what I am striving for in that the text is not analysed or identified by the user, so it does nothing to further their knowledge of the analysis process, and does not generate the trees in the format or syntax lined out in the core text I am using [4].

2.2 Technical material

Stackoverflow [5] proved very useful in every stage of the project, as I encountered issues in any areas. Problems I came across in my project were obviously quite common, as I usually found a solution to high level and low level programming problems on this website. Through the use of this website, and websites like it, I was able to improve my technical competence in areas from client-side web development (HTML, Js, CSS), and in two forms of server side scripting, as well as SQL. In searching for inspiration on ways to implement my project, I also familiarised myself with the different frameworks linking these two aspects of the one service together.

Although one of the architectures produced is full stack Js, it is not MEAN, as I decided not to use MongoDB, and instead chose saving to a JSON file for easy and quick data transit by the user.

The annotator website [6] was utilized in working with the Annotator.js library, where there are links to the documentation of the library, examples, plugins, and general help. The tagging library I used was taggle [7]. This also included examples, documentation and advice for its use.

I had not used Node.js before, and so obtained advice and instruction on its use from various sources, including Stackoverflow, as mentioned above. Much useful information was also gleaned from the URLs referenced at the end of this report [8] [9]

The information from these sites were cross referenced and used in tandem with one another to achieve the desired result.

Chapter three: The Problem

3.1 Problem Introduction

The problem at the very core of this project is how to convert information about a piece of text, specified by the user, to a visual realisation in the form of a tree. This visual realisation must not only preserve that information, but build upon it through the use of knowledge subconsciously dictated by the user, such as its position in the overall text. Once this has been overcome, the problem then shifts to how best to get the user to input the data, and the most effective way to display the trees.

3.2 Project UML documentation

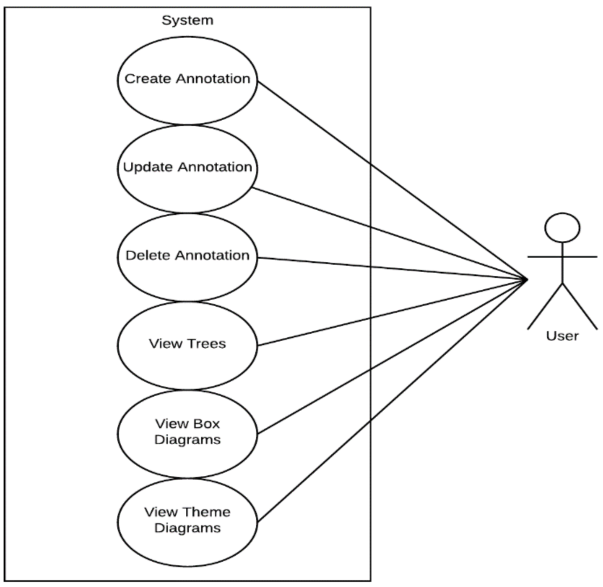


Fig. 3.1 - Use Case Diagram

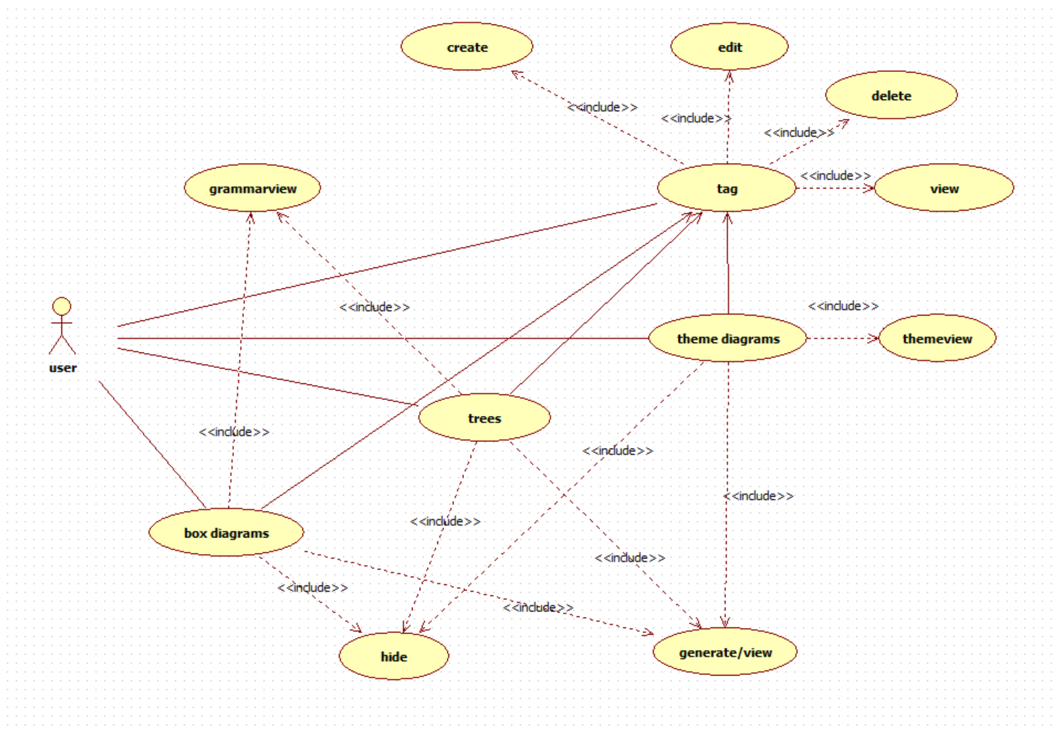


Fig. 3.2 - UML Diagram

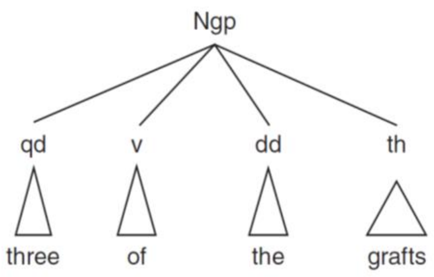
3.3 Problem analysis

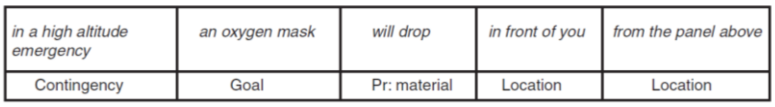
There are multiple problems to be addressed in this project, the most important of which to solve is how to develop some service that allows completion of the main task; generation of the tree diagrams from user tagged text. It would not make sense starting development from scratch, as it would be a waste to redo what has already been done and is available free to the public.

A proper analysis had to be done before any third party libraries could be chosen, so the main task was broken down into precise and well defined requirements. Users must be able to identify and label text according to the principles of systemic functional grammar analysis. They should not be constrained by having to tag the text in any particular order, and any analysis done should be very easily extended, amended or reversed as required. Inputs from the user should be limited to predefined labels that are packaged with the application, to eliminate any spelling mistakes or deviations from homogeneity. The list of available labels should be easily visible to users, making it clear what is allowed.

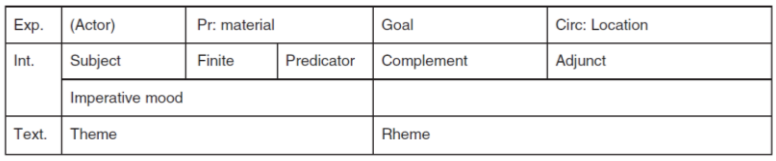
Any relevant data must be collected and stored during this phase, to be used in generation of tree structure at a later time. Some of the data needed would include the length of the tagged text, the position of the text in the document, and the label(s) specified by the user. Some kind of storage will be needed to save data; indefinitely if required, and maintaining the information without loss or corruption. Desired data must be readily and easily retrievable.

Once the format, method of collection and storage of the data have been finalized, it must be utilized to generate the SFG tree diagram. The format of the data must be chosen carefully to make this process as intuitive and transparent as possible, to leave less work for myself, and to allow simple expansion of





in a high altitude emergency	*an oxygen mask*	*will drop*	*in front of you*	*from the panel above*
Contingency	Goal	Pr: material	Location	Location



Exp.	(Actor)	Pr: material		Goal	Circ: Location
Int.	Subject	Finite	Predicator	Complement	Adjunct
	Imperative mood				
Text.	Theme			Rheme	

Fig. 3.2 - Example of a Syntax Tree, a Box Diagram (above), and Theme Diagram (below) [4]

the application in the future. An organization of the data will take place, placing an order and hierarchy upon it, using just the tagging information.

The last aspect of the problem is to find an appropriate and flexible library that allows generation of tree diagrams from some appropriately encoded JSON, which will be outputted by the previous task. It would be most useful if this library natively allowed the style of tree required, and had the utility of adding custom html to respective nodes. Relevant tree diagrams must be sent to the page and displayed to the user. As mentioned before, throughout these processes, any user interaction should be as intuitive and easy to use as possible, with an attractive and simple interface.

On completion of the main task, support would be then implemented for box diagrams, which could, in principle, be generated from the same data as was used in the tree generation, cutting down on user required input, time usage and computation time within the application.

Support for theme analysis could also be added, which would use the same text, but tagged separately. Whichever design decided upon for the functional analysis would be replicated for thematic analysis, which should leave it easier to implement. Relevant diagrams would be created from the tagged data, and displayed to the user, just as in the case for the tree diagrams.

If implementation of these core features is successful and time allowing I will investigate various deployment options of the application, ensuring it is as available and accessible to as many people as possible, developers and consumers alike. This analysis will lie in how the tags are stored, whether locally or remotely, and how easy or difficult it is made for users in both respective cases.

Chapter four: The Solution

4.1 Analytical Work

Most of the analytical work carried out for this project involved an in-depth inquiry into the nature and type of data used by each prospective library, and fabricating an algorithm to make the conversions mentioned in the previous chapter. The investigations into the data concluded that JSON was the best format for storage and manipulation, as it was supported natively across all libraries, and facilitated ease of use in posting to the various servers.

As the application was to be available to as many people as possible, various server architectures and frameworks were looked at. Since the Js framework and architectures concerning the use of php and sql were most popular, and widely used, I focussed my attention on those initially.

After rudimentary back ends had been built, my attention turned toward ways of facilitating users that don't have connection to the internet, or possibly even a web browser. This led to the decision to put development efforts into an offline variant. The Electron software framework, by Atom.io, was brought to my attention by my supervisor, who recommended research into whether or not it would be feasible as a tool to bring the application to desktop users.

Past knowledge I had acquired in CS210 and CS211 proved most useful in the writing of code to construct and traverse the tree objects within the project. I knew from the specifications of the project that my solution would not be computationally expensive, it being a web application, so no resources were spared in the early construction phases.

4.2 High Level

As mentioned in the summary of the problem, I implemented an XAMPP server on my machine, which was used for initial backend work and storage. The latter was done through phpMyAdmin and a MySQL database.

What my code needed to be able to do was take in user input, allow the user control of that input, in terms of editing or deleting it, whilst displaying user annotations to the screen on hover. I set up annotator to work in conjunction with taggle to achieve this, ensuring lossless data handling between the two. When working with the XAMPP installation, methods were written to handle communication with the php backend, and likewise for the Node variant.

Each function of the application instigated by the user was mapped to a button, and a respective method was written to handle each with jQuery. For example; When the user required the display of a Syntax Tree, they click a button, which calls a method, loading the latest data straight from the respective database, making the necessary computations to generate the tree from that data, and finally displaying the end result to the user. The process for theme and box diagrams was similar, and reused the same methods where possible, to cut computation time and repetition.

4.3 Low Level

One of the most important aspects of the project was the ability to save tagged text, in this case, through the media of an annotation. Annotator.js allows storage through a number of different plugins, none of which suited my particular needs. I came across a project on GitHub that used annotator in this way, in conjunction with a php backend. I rewrote and repurposed this code for use with my application, altering it as required for the differing versions. With the php backend, it posts to the php files, with the Node backend, it posts to the Node Server, and with the offline and desktop versions, it handles saved data within the javascript variables on that page.

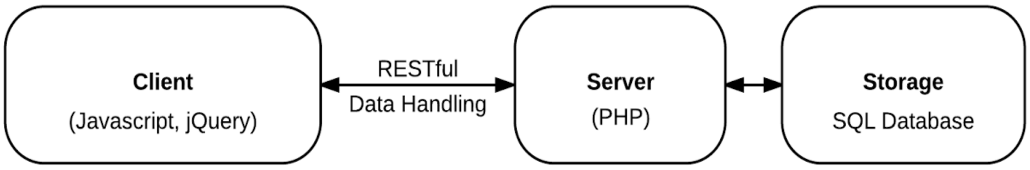


Fig. 4.1 - XAMPP Architecture Diagram

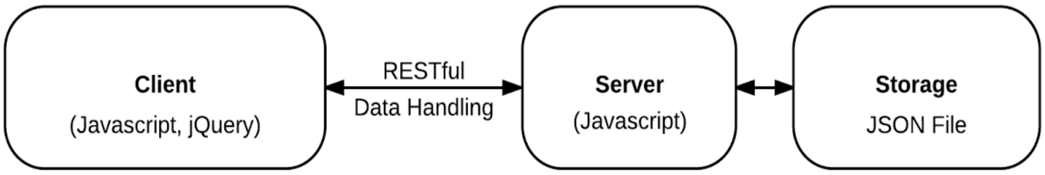


Fig. 4.2 - Full Js implementation Architecture Diagram

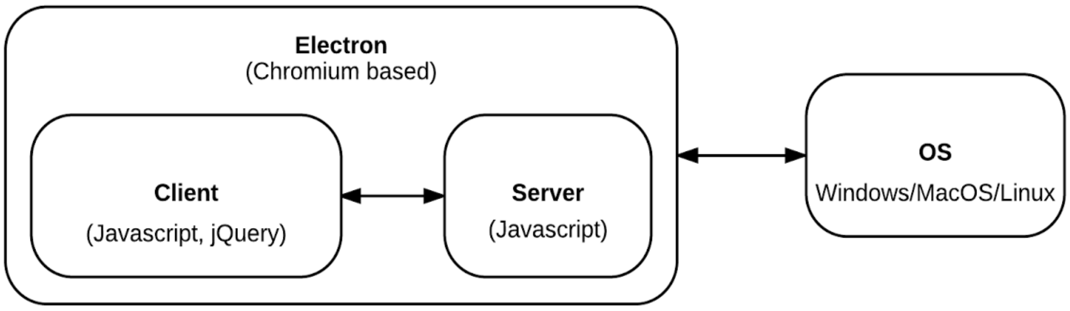


Fig. 4.3 - Electron Architecture Diagram

I used JSON across all platforms, due to its fast, compact and convenient serialisation of data. The ease with which it could be converted to a JSON string was perfect for sending to the sql database, and subsequent storage. JSON was already natively used by Annotator, Taggle and Treant, which made it the obvious choice without even considering the mentioned benefits.

The largest problem to be overcome was how to use one form of abstract data structure (annotations supplied by Annotator) to another (syntax trees required by application). A useful feature of Annotator is to record all available information about the annotated text, including that which was of particular interest to me, the length of the annotated phrase, the exact start and end positions of the annotation, any HTML tags within which it resides, and of course, the user specified text itself.

The first task in the data conversion, was to create a node object generated from each annotation, maintaining the integrity of all attributes just mentioned, but in a more intuitive manner for the tree library being used. I then cycled through all nodes, sorting them by the length of the annotation, and then cycling through them, assigning nodes with ranges that are proper subsets of other nodes as the children of the latter. A rudimentary tree structure was generated in this way, and this could be refined or adjusted as I wished.

Each node's children were then ordered according to their position in the document, leaving the tree structure in the correct form, but lacking in syntax. This tree object was accessed by all features of the applications, whether to generate the syntax trees, box diagrams or theme diagrams. In the interests of the DRY (Don't Repeat Yourself) coding mantra, I tried to minimise the code I had to write. Since I was dealing with conversion between so many abstract data structures, code was constantly in danger of becoming unwieldy, unreadable, or any linear combination of the two.

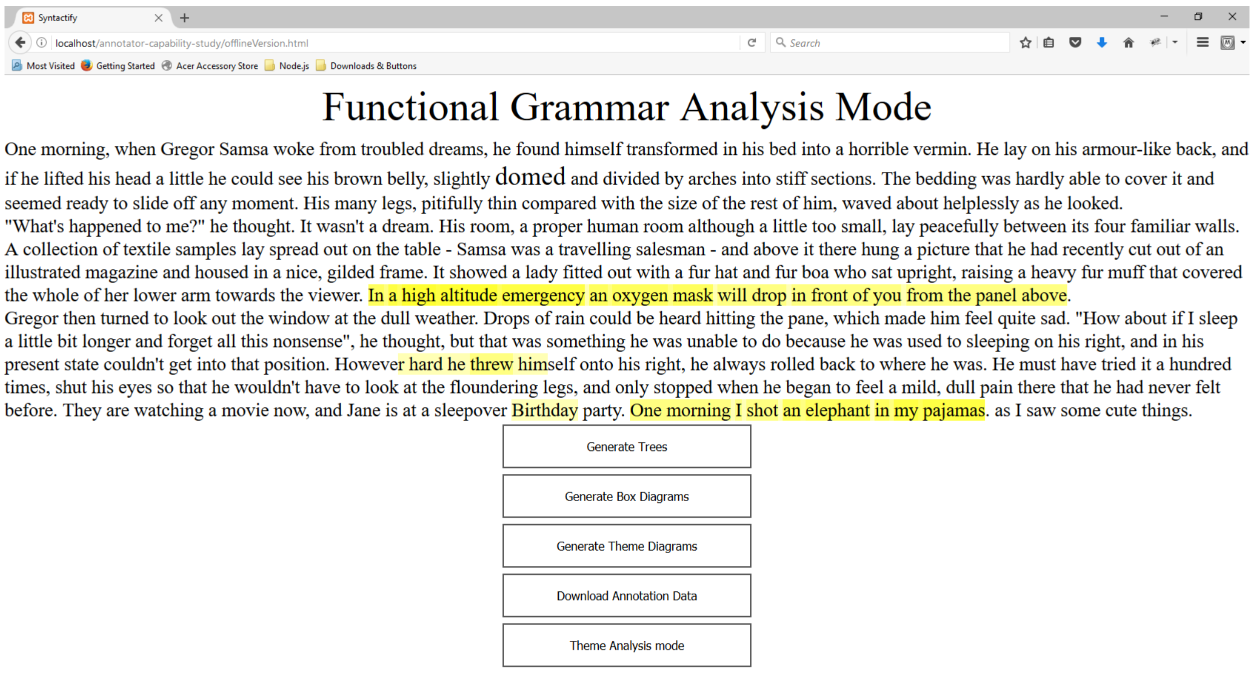
For the generation of the syntax trees, I wrote several methods and called them in succession, each altering the tree slightly, making it more like the end goal. I added a specification to the user instruction requiring the user to input all tags for a particular phrase in one annotation, and in the correct order.

The tree library chosen allows the insertion of custom HTML tags, which facilitated the customisation necessary of each node to meet the visual specification of the original material. To match the required style, I built up the HTML tags piece by piece as strings, and added them as attributes to the node objects, following the Treant API.

SVG tags are natively used by Treant when displaying tree diagrams and added to the page directly after the initialisation phase. This was another attribute in the favour of this library, as SVG are the web equivalent of vector graphics, and far superior in terms of flexibility and scaling compared to HTML canvas methods.

With the generation of box and theme diagrams, I concocted a specification for user input, making efforts to keep it as simple and intuitive as possible, that would minimise confusion in use, as well as the code I would have to devise. The particulars for this are described below.

4.4 Implementation

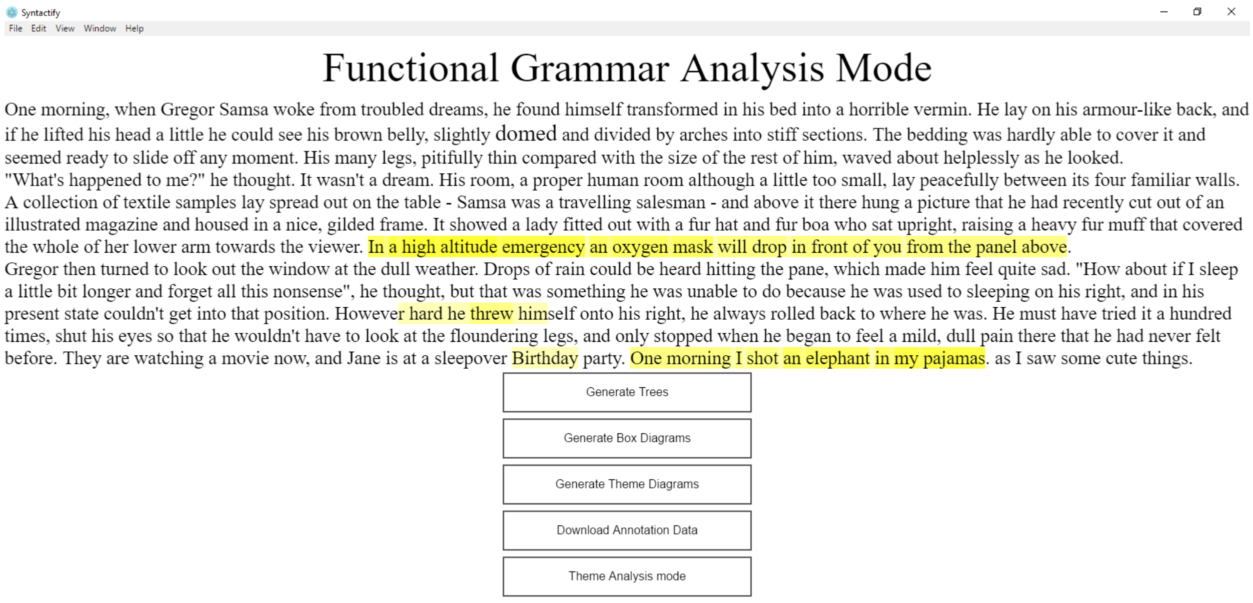


Functional Grammar Analysis Mode

One morning, when Gregor Samsa woke from troubled dreams, he found himself transformed in his bed into a horrible vermin. He lay on his armour-like back, and if he lifted his head a little he could see his brown belly, slightly domed and divided by arches into stiff sections. The bedding was hardly able to cover it and seemed ready to slide off any moment. His many legs, pitifully thin compared with the size of the rest of him, waved about helplessly as he looked. "What's happened to me?" he thought. It wasn't a dream. His room, a proper human room although a little too small, lay peacefully between its four familiar walls. A collection of textile samples lay spread out on the table - Samsa was a travelling salesman - and above it there hung a picture that he had recently cut out of an illustrated magazine and housed in a nice, gilded frame. It showed a lady fitted out with a fur hat and fur boa who sat upright, raising a heavy fur muff that covered the whole of her lower arm towards the viewer. In a high altitude emergency an oxygen mask will drop in front of you from the panel above.
Gregor then turned to look out the window at the dull weather. Drops of rain could be heard hitting the pane, which made him feel quite sad. "How about if I sleep a little bit longer and forget all this nonsense", he thought, but that was something he was unable to do because he was used to sleeping on his right, and in his present state couldn't get into that position. However hard he threw himself onto his right, he always rolled back to where he was. He must have tried it a hundred times, shut his eyes so that he wouldn't have to look at the floundering legs, and only stopped when he began to feel a mild, dull pain there that he had never felt before. They are watching a movie now, and Jane is at a sleepover Birthday party. One morning I shot an elephant in my pajamas. as I saw some cute things.

Fig. 4.4 - Application working in browser

Once a text has been uploaded, the user then selects whether they will start analysis from scratch, or upload pre-existing annotation data, in JSON format. Once an option has been selected, the application makes the correct respective preparations, displaying uploaded data, or initialising from scratch.



Functional Grammar Analysis Mode

One morning, when Gregor Samsa woke from troubled dreams, he found himself transformed in his bed into a horrible vermin. He lay on his armour-like back, and if he lifted his head a little he could see his brown belly, slightly domed and divided by arches into stiff sections. The bedding was hardly able to cover it and seemed ready to slide off any moment. His many legs, pitifully thin compared with the size of the rest of him, waved about helplessly as he looked. "What's happened to me?" he thought. It wasn't a dream. His room, a proper human room although a little too small, lay peacefully between its four familiar walls. A collection of textile samples lay spread out on the table - Samsa was a travelling salesman - and above it there hung a picture that he had recently cut out of an illustrated magazine and housed in a nice, gilded frame. It showed a lady fitted out with a fur hat and fur boa who sat upright, raising a heavy fur muff that covered the whole of her lower arm towards the viewer. In a high altitude emergency an oxygen mask will drop in front of you from the panel above.
Gregor then turned to look out the window at the dull weather. Drops of rain could be heard hitting the pane, which made him feel quite sad. "How about if I sleep a little bit longer and forget all this nonsense", he thought, but that was something he was unable to do because he was used to sleeping on his right, and in his present state couldn't get into that position. However hard he threw himself onto his right, he always rolled back to where he was. He must have tried it a hundred times, shut his eyes so that he wouldn't have to look at the floundering legs, and only stopped when he began to feel a mild, dull pain there that he had never felt before. They are watching a movie now, and Jane is at a sleepover Birthday party. One morning I shot an elephant in my pajamas. as I saw some cute things.

Fig. 4.5 - Application running in windows through Electron

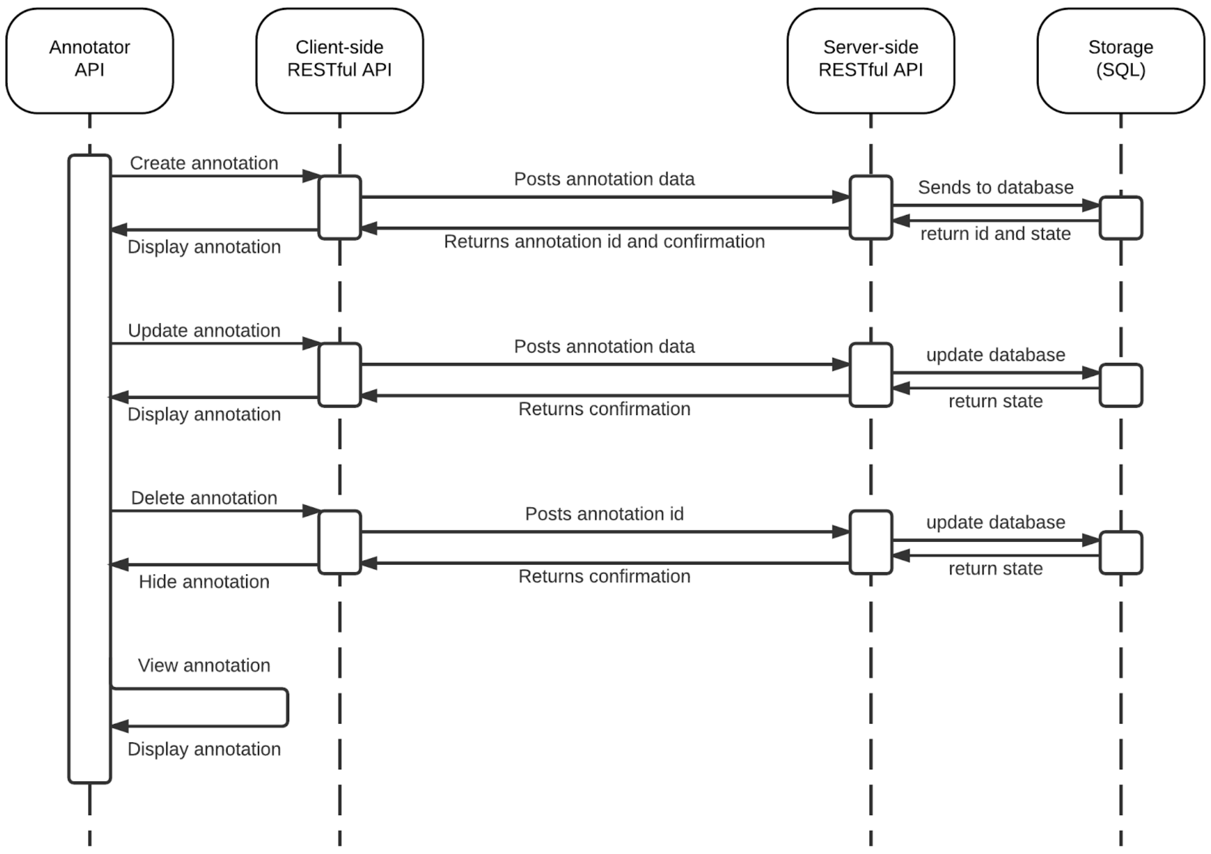


Fig. 4.6 - Interaction Sequence Diagram for annotation handling in the XAMPP variant

When either the system has been provided with previous annotation data, or initialised from scratch, the user can then add new annotations as well as edit or delete any that have been uploaded. The auto completion feature is available to the user at this stage.

After the text has been tagged, the user can generate syntax trees, box diagrams and theme diagrams with the click of a button.

If the user wishes to analyse the theme of a text, they select the button “Theme Analysis Mode”, which instigates a transition to a different instance of annotation catering specifically to investigate the theme. A separate view and annotation method was necessary to facilitate support of the theme, as there was not enough information supplied by the annotations used in generating syntax trees and box diagrams.

As can be seen from Fig. 4.3 and 4.4, the user interface was normalised across all variants, each version is identical in function, differing only in architecture, as outlined in the previous section. Data handling across all types was standardised so that data exported as JSON from the sql database could be uploaded to the offline and desktop versions, or placed as the file written to by the Node version. Likewise, data downloaded from the offline or desktop variants can be used by the Node version. Sending this data back to the sql database involved converting the JSON to CSV, but is still straightforward.

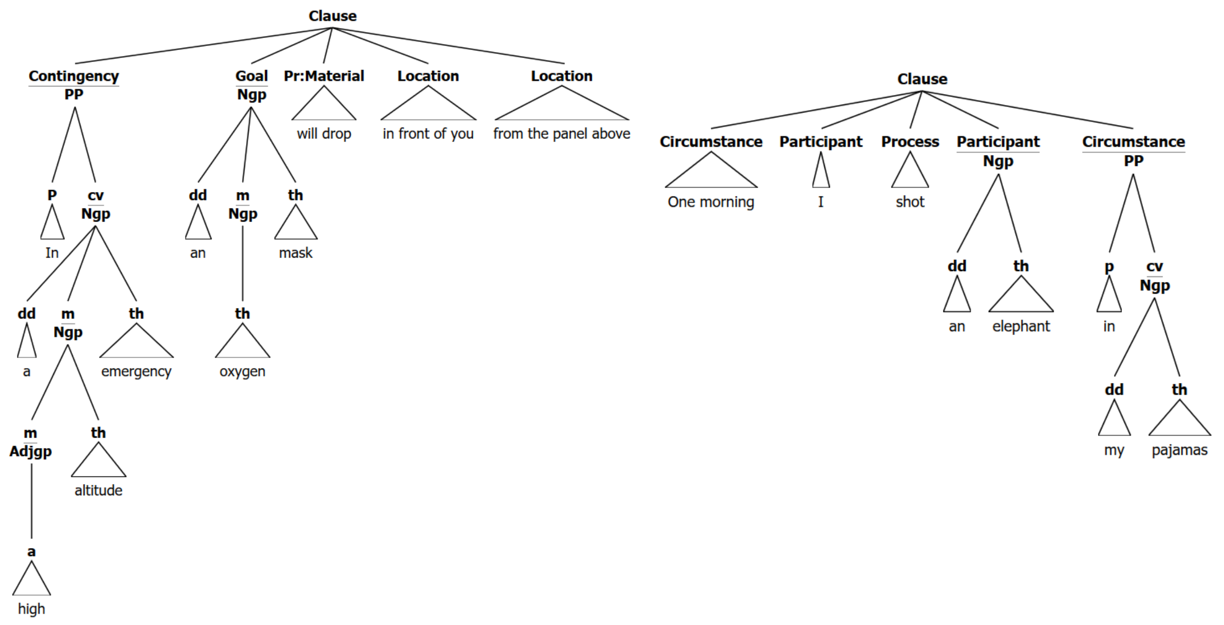
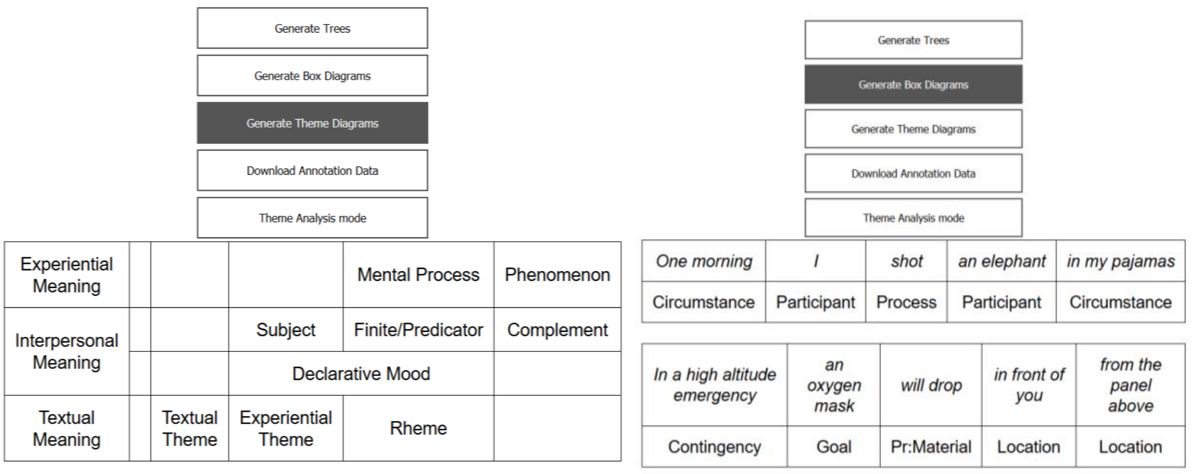


Fig. 4.7 - Syntax trees generated by the application



Experiential Meaning				Mental Process	Phenomenon
Interpersonal Meaning			Subject	Finite/Predicator	Complement
			Declarative Mood		
Textual Meaning		Textual Theme	Experiential Theme	Rheme	

One morning	*I*	*shot*	*an elephant*	*in my pajamas*
Circumstance	Participant	Process	Participant	Circumstance

In a high altitude emergency	*an oxygen mask*	*will drop*	*in front of you*	*from the panel above*
Contingency	Goal	Pr:Material	Location	Location

Fig. 4.8 - Theme and box diagrams generated by application

Chapter five: Evaluation

5.1 Solution Verification

As mentioned previously, the software will be verified by a comparison between the source diagrams and the diagrams generated by the application.

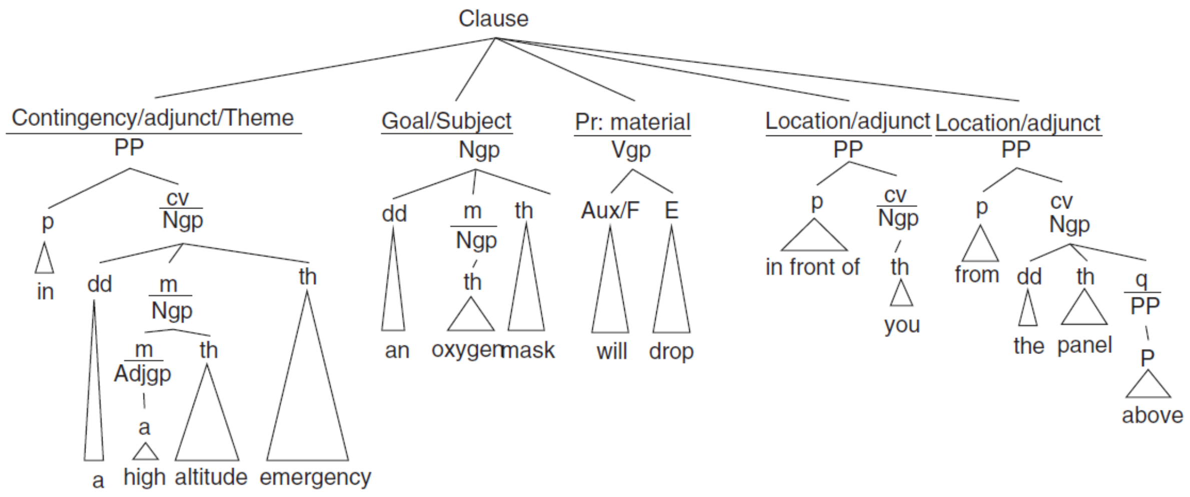
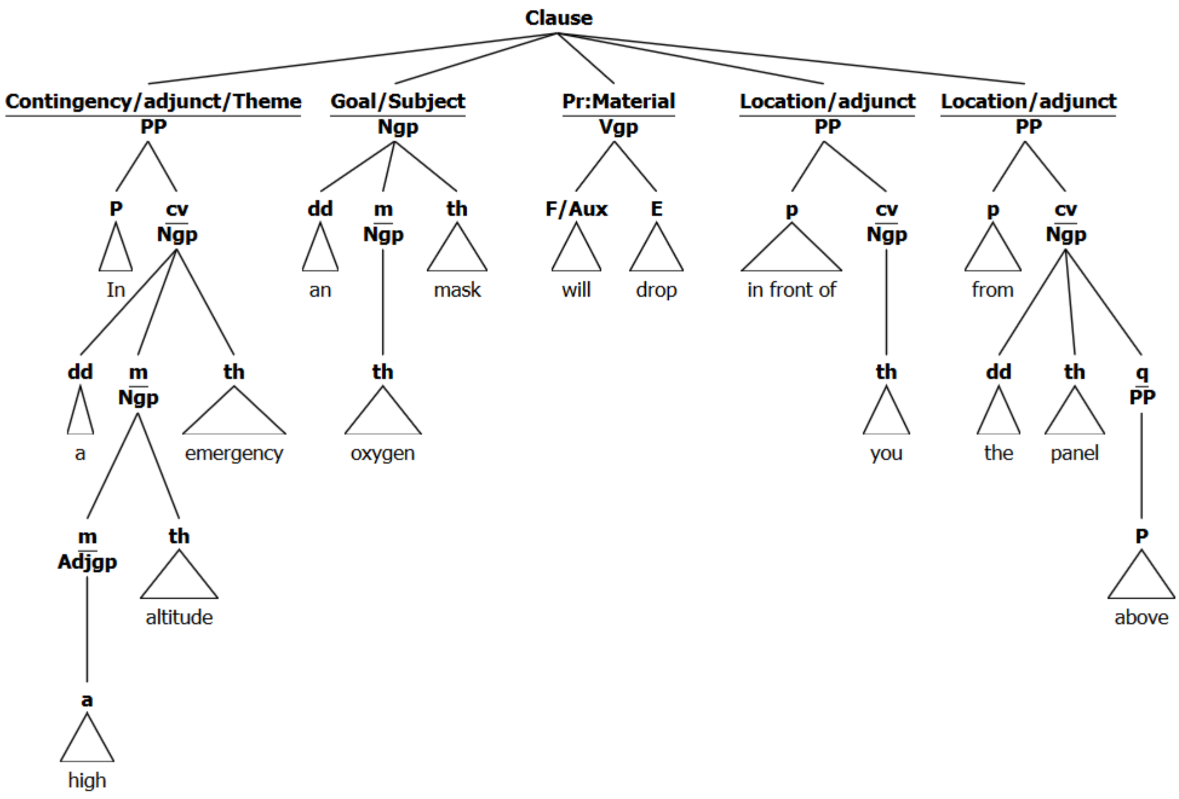


Fig 5.1 / 5.2 - Original syntax tree (above) [4] */ and generated version (below)*



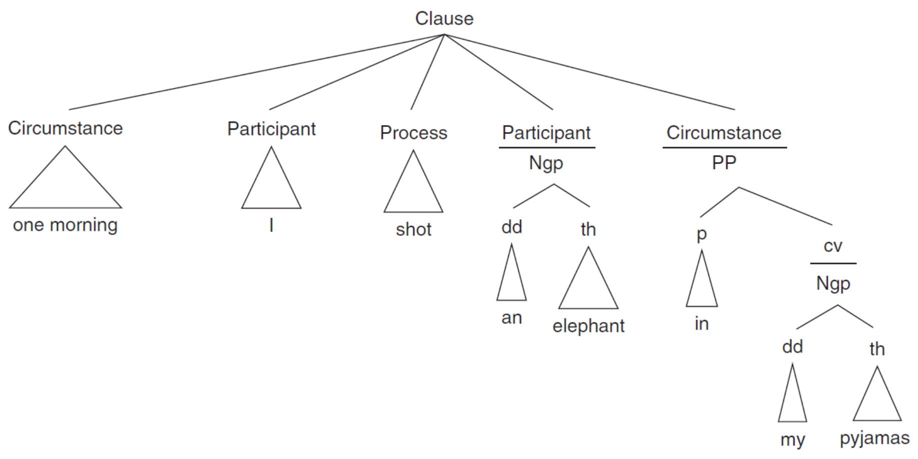
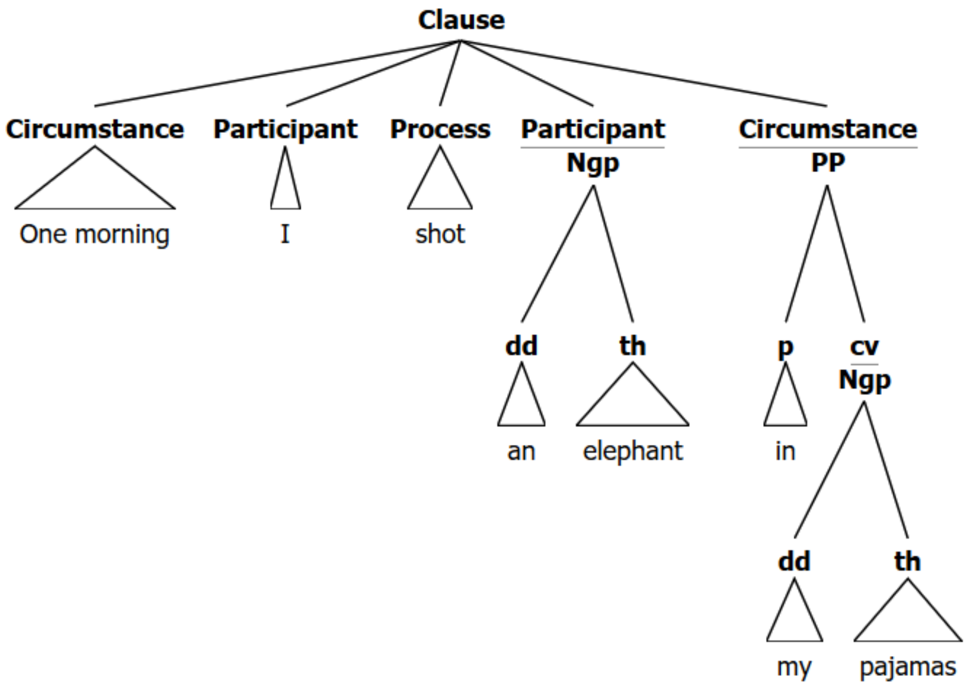


Fig 5.3 / 5.4 - Original syntax tree (above) [4] */ and generated version (below)*



As can easily be seen from the above, my diagrams preserve the syntactical and structural integrity shown in the original example, and are in fact superior in terms of proportion and symmetry. If the end user wishes to alter the proportionality of the tree, it is quite easy to do within the main.js file. The distance between levels and siblings can be adjusted pixel by pixel until the desired result is achieved.

5.2 Software Design Verification

Mozilla Firefox was used as the main testing tool, due to its native allowance of element inspection, and selection of developer plugins. Through the use of Firebug for Firefox, I was able to constantly verify the functionality of the server frameworks, and ensure any and all post requests made by the client were correct. When using the sql database through php, I checked its contents after each creation and edit of an annotation, as a way of investigating the effectiveness of my code.

Whilst using the Node installation, which wrote data directly to a JSON file, I periodically queried its contents, including when they were altered or appended in any way. Just as with the sql database, this allowed me to regulate the data and confirm its integrity as well as verify the function of my code.

On the completion of the application, I introduced it to four of my peers in the computer lab to get their opinion on the User Interface. They reached a consensus in that the method used in tagging the software was indeed intuitive and sensible for tree generation, and that the method of annotating the text was a delicate, but forgiving, process.

5.3 Software Verification

Great emphasis was placed upon testing in the project, with code regularly being tested even as it was being written. After any piece of code was completed I carried out unit testing, showing each individual part achieved its goal in terms of functionality and requirements, through isolation and specific test cases. Some limitations of unit testing are that it cannot catch each and every bug in an application, but since there was a comparatively low limit to the scenarios my code would be exposed to, these weaknesses were not a worry.

On the completion of a method, and its basic unit testing, it was then amalgamated with the complete application at that respective time, and put through integration testing. Since unit testing was carried out first, this method is called Bottom-up Integration Testing. Seeing that an agile methodology was used for development, and consequently, that more features were added on each iteration, some rudimentary regression testing had to be performed. This was to ensure no negative effects were seen in the performance of any of the other code modules implemented earlier.

Stress testing was also carried out, but to a much lesser extent than the other forms. I was not particularly concerned about the application maintaining its composure and integrity under stress, as it performed to specification on the largest tree I could find in the core text [4], and if for any reason it were to crash or fail, it can be reset with a refresh of the page or browser. Animations using jQuery were tested in this way, to ensure adequate performance for an impatient or eager user. Performance testing came into play here, in confirming animations were executed synchronously in all conditions.

The final hurdle was functional testing, the culmination of testing in the project, and arguably the most important. I divided the system as a whole into slices of functionality, testing each individually, before combining them in different combinations. Techniques used include the aforementioned regression testing, as well as smoke testing. Smoke testing (also known as build verification testing) quickly determines whether or not the system is so broken, as to make further testing unproductive.

All testing carried out was manual, as it was not worthwhile writing automated scripts due to the complexity and uniqueness of test cases.

5.4 Software Validation

The final step in the evaluation was validating the application, ascertaining whether or not the system complies with the specified requirements and performs core functions adequately and satisfactorily. When comparing the software to the original specifications, it is seen that all of the core features are encompassed within it, as well as all of the desirables.

Chapter six: Conclusion

6.1 Contribution to the state-of-the-art

As mentioned previously in the motivation section of the report, grammar analysis is essential in the conception and manufacture of software that can decode the meaning of speech, both oral and written. Systemic grammar is one of the most prominent forms, as it can be used to analyse text, making accurate inferences and predictions as to its context, mood, theme, rheme, textual meaning, interpersonal meaning and experiential meaning.

This said, it is crucially important to have specialists in this field, allowing the development of such software. This software is intended as a pedagogical tool to assist in the visualisation and understanding of the process of Systemic Functional Analysis. By making this process more accessible and intuitive for the end user, the application provides a useful tool with which they can simplify this process and facilitate the introduction of these fields to budding linguists and novices alike.

6.2 Results discussion

Although satisfactory results were achieved as an end product of the project, there is a type of syntax tree inherently impossible to generate using my methods.

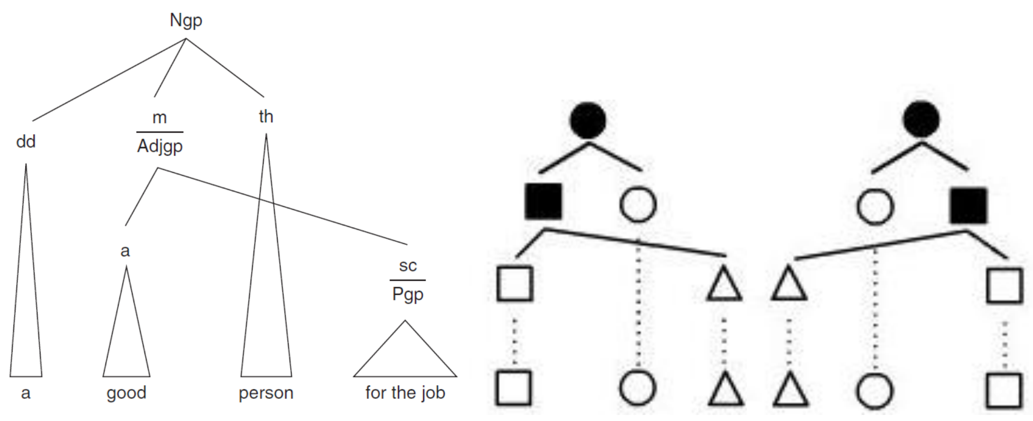


Fig. 6.1 - Examples of trees with discontinuous scope [4] [10]

This was due to a couple of factors, most prominently, the tree library that was used. Even though it was the most flexible and feature rich library I could find, it did not have support for discontinuity in structure. I went to the lengths of contacting the developer through the medium of GitHub in ascertaining this. If it were the case that this type of structure was supported, there would still have to be a full overhaul in annotation design and use to facilitate the generation.

A new tree generation library would have to be coded to allow support for these diagrams, or at the least large changes would have to be made to the existing tree generation library. I spent some time analysing the source code, but could not fabricate a non-trivial path toward a solution.

6.3 Project Approach

There are a few things I would have done differently if starting this project from scratch again. In particular, I would try to generalise the program such that it could be used with future versions of Annotator, as my version is specifically built for Annotator v1.2.7. This was necessary as it was the latest version to provide the full source code, all versions since have been minimised (a process that systematically replaces all variable and method names with the shortest legal options available) leaving the code human unreadable, and thus inextensible.

It was unavoidable altering the source code in the way that I did, to implement taggle. If I were to adopt an approach within which taggle was not used, it would have been possible, but user input would have to be carefully formulated and entered, with no room for error in syntax. This would have made the software most user unfriendly, and potentially turn people off using it.

6.4 Future Work

There are many possible avenues this application could open up in the field of functional linguistics. I have made great efforts to ensure my code is easily understandable and extensible, by inundating it with comments, and maintaining rigorous and informative documentation of each method within it. I am aware that what I have built may be used as a base for future work in analysis tools for linguistics, and I have reflected this in the manner of development.

All that is required to deploy this application is to host the page upon which it is implemented. The service runs entirely within one html page, making it light and portable. Once this application is made publicly available, my hope is that it will be used and extended into a full scale analysis tool, supporting work with multiple texts simultaneously, and allowing specific clause selection, editing and dissection.

References

[1] M. Halliday, “System and function in language, selected papers,” London, Oxford University Press, 1976.

[2] D. Butt, R. Fahey, S. Feez, S. Spinks and C. Yallop, Using Functional Grammar: An Explorer's Guide, Sydney: Macquarie University, National Centre for English Language Teaching and Research, 2003.

[3] Z. Feng, “Functional Grammar and Its Implications for English Teaching and Learning,” *English Language and Teaching; Vol. 6, No. 10,* p. 9, 30 July 2013.

[4] L. Fontaine, Analysing English Grammar - A Systemic Functional Introduction, Cambridge: Cambridge University Press, 2013.

[5] “StackOverFlow,” [Online]. Available: https://stackoverflow.com/. [Accessed 16 03 2017].

[6] “Home - Annotator - Annotating the Web,” [Online]. Available: http://annotatorjs.org/. [Accessed 16 03 2017].

[7] S. Coker, “Taggle.js | Sean Coker,” [Online]. Available: https://sean.is/poppin/tags. [Accessed 16 03 2017].

[8] “Learn Node.js,” [Online]. Available: https://www.tutorialspoint.com/nodejs/. [Accessed 16 03 2017].

[9] “Felix's Node.js Beginners Guide,” [Online]. Available: http://nodeguide.com/beginner.html. [Accessed 16 03 2017].

[10] “Wikipedia,” [Online]. Available: https://en.wikipedia.org/. [Accessed 16 03 2017].