Kleene's Theorem – A Web-Based Application

Samuel Heersink

Honours Project Report

12/17/18

Table of Contents

Introduction

Introduction to Formal Languages was one of my favourite classes of my undergraduate degree, and the concept of formal structures in general has been a subject of continued interest to me. Thus, when asked to choose a subject for my honours project, I knew exactly where to turn. Thomas Tran, who taught me the Formal Languages class, was extremely helpful in providing a fully fleshed-out idea for an honours project: a teaching tool that demonstrates some core concepts of formal languages through a web-based application. Professor Tran laid out the features and goals of this project, and I set to work on achieving them.

This report covers the conception, development and execution of this project over the duration of the past semester. After providing some background information on the formal structures relevant to the project, I give an overview of the project goals before elaborating on the structure and functionality of the application. Afterwards I share some reflections on the development process and suggest some potential next steps should there be continued development on the application.

I am quite satisfied with my achievements on this project and I hope that the insights provided in this report are conducive to understanding the application's usage and structure.

Background Information

A regular expression is an expression that matches strings possessing a particular pattern. We call this set of strings the "language" of the expression. The expression can be made up of any number of characters from the alphabet used by these strings, as well as a small collection of operators: parentheses "(" and ")" for operational precedence, "+" for the union of two possibilities, "λ" for the empty string, and "*" for the Kleene closure of an expression: the repetition of the expression any number of times. The regular expression representing the language of strings with at least two occurrences of the letter "a" using the alphabet {a, b} is as follows:

b*ab*a(a+b)*

A transition graph is a collection of states and transitions that can be used to test whether input strings match a particular pattern. The transition graph is made up of an alphabet of characters that it recognizes, at least one start state, and any number of intermediate and final states. The graph also includes transitions between states, labelled with regular expressions. The pattern of strings recognized by a transition graph is determined by the graph's structure. To check a given string, choose a start state in the graph. Starting at the beginning of the string, remove characters that match the label of a chosen transition. Continue taking transitions and removing characters this way until you reach the end of the string. If the string terminates in a final state, then the transition is accepted. Otherwise, the string is rejected. The

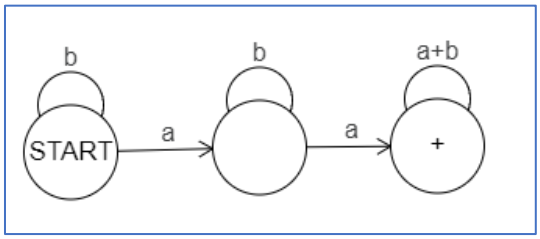

Figure 1: a transition graph representing the language of all strings with at least two "a"s

transition graph representing all strings with at least two occurrences of the letter “a” using the alphabet {a, b} is shown in Fig 1.

The transition graph is an extension of a third structure, the finite automaton. A finite automaton is like a transition graph in that it is made up of states and labelled transitions but differs from the transition graph in that transitions are labelled with single characters instead of full expressions, it possesses only one start state, and each state must have exactly one outgoing transition for each character in the language. With these changes, the possibility of choice is removed from the traversal of the graph – checking the validity of a string with a finite automaton becomes a deterministic task. The finite automaton representing the same pattern of strings as fig(X) is almost identical: the loop on the third state would simply be replaced by one loop for “a” and one for “b”.

Kleene’s theorem states that these three structures are equivalent, and that any language described by a regular expression has a corresponding deterministic finite automaton and transition graph that represent the same language. [1]

Project Goals

The project proposed to me by professor Thomas Tran was the development of a web-based application that demonstrates the properties of Kleene’s Theorem of finite automata. This application should allow users to create their own transition graphs by drawing on an HTML canvas. These graphs are verified for validity and can be saved as PNG images to be embedded in documents. In addition, the software should demonstrate part 2 of Kleene’s theorem, the equivalence of transition graphs and regular expressions. This may be demonstrated by allowing users to convert the transition graph they created into a regular expression by working their way through the constructive algorithm as described by Kleene. An important feature of the application is that it is instructive in nature: the application should not directly output the equivalent regular expression but should instead give the user a chance to learn about the individual steps of the constructive algorithm.

An additional phase of development was proposed, which is the demonstration of part 3 of Kleene’s theorem: the equivalence of regular expressions and finite automata. This can be demonstrated by generating finite automata representing the concatenation, union or Kleene closures of finite automata provided by the user, coinciding with the union, concatenation or Kleene closures of the corresponding regular expressions. In this way it is shown that the construction of finite automata can be performed in the same way as the construction of regular expressions.

Overview of Application Structure and Usage

Class Structure

A critical aspect of the application design was the underlying structure of the transition graph. Maintaining consistency between the structures as stored in memory and the graph as displayed on the canvas is very important for ensuring transparent execution and avoiding graphical misbehaviour.

Representing any kind of structure within a program begins with the class definitions. Fig 2 shows the UML class diagram of the three core classes used by the application, displaying each class' variables and methods. These classes and their methods are used for the basic manipulation of the graph and are called by the palette actions operated by the user. Each *Transition* and *State* have their own references to the canvas context so that they can draw themselves when called by the graph's draw method. Each *Transition* and *State* also have their own unique ID, generated on their creation by a global static method that increments a counter and returns its value as an ID. Each state stores a list of the states that can be reached directly from that state (*precedes*), as well as a list of the states that can reach that state directly (*follows*).

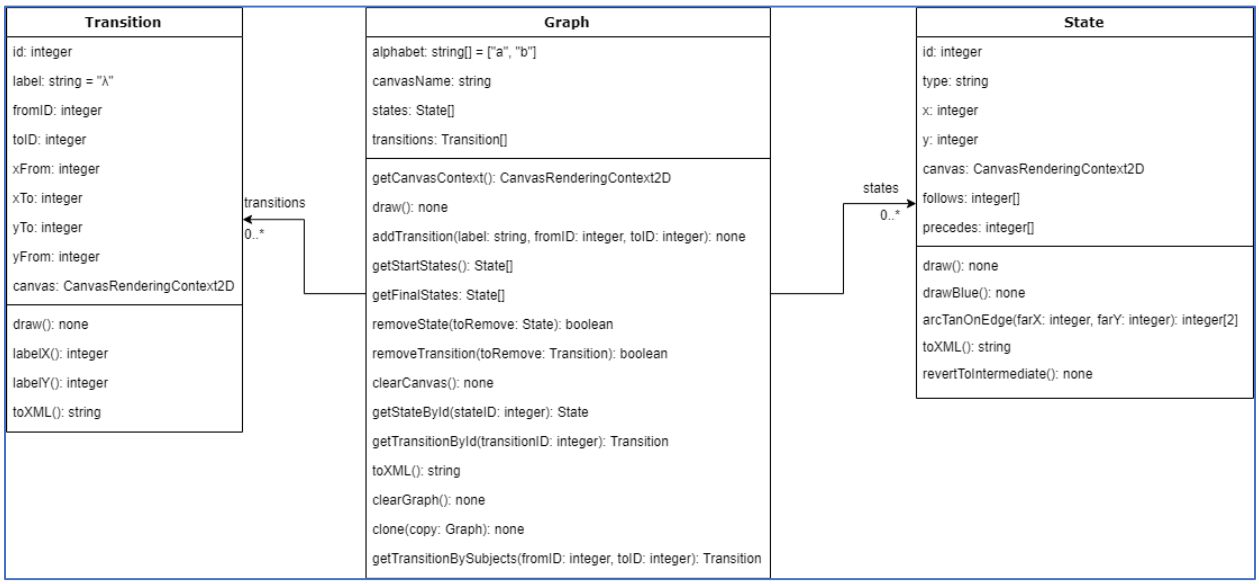

Figure 2: The class diagram of the structures that make up the transition graph as stored in the application memory

Palette Function Usage

The palette is the collection of buttons available to the user for manipulating the graph. The palette functions called by these buttons are top-level graph manipulation functions that in turn make calls to class methods and modify the underlying graph structure. Because some of these operations require multiple actions from the user, the *PaletteController* class is required to monitor the state of operations, transition between these states, and prevent actions from interfering with each other.

Clicking the alphabet button opens an editable text box containing the current alphabet, separated by commas. Clicking a second time applies a regular expression to check that the textbox content contains exclusively letters separated by commas and spaces and saves the new alphabet if successful.

Adding states of various types is a simple task: clicking the button allows the user to choose a location on the canvas and an optional label for the new state. If the selected location is not too close to another state, a new state with the specified type, location and label is added to the graph.

Creating a transition is the action that requires the most support. The user clicks the button, then chooses a state for the transition to begin and an expression for the transition's label. That transition is redrawn with a blue border, and the user is prompted to choose a destination for the transition. Once the destination is chosen, the program calculates the point on the edge of each state that is closest to the other using the arctangent of the vertical and horizontal distances between the two states' centers. A line is drawn between each state's point, and an arrow is added to the destination end of the line. Finally, the transition is drawn, and the label is drawn at the center of the transition line. This involved process is demonstrated with an activity diagram in Fig. 3.

Deleting an element from the graph is a matter of comparing the clicked location's coordinates with the coordinates of each state and transition in the graph. When a state is deleted, all transitions connected to it are deleted. When a transition is deleted, each state previously connected to it has its *follows* and *precedes* lists updated accordingly.

Another action requiring a good deal of support is *undo*, which reverts the graph to how it was prior to the last edit. A "checkpoint" copy of the graph as last seen is stored in memory along with the main graph object, and when the undo action is performed the main object is replaced with the checkpoint. To enable this, the *Graph* class is equipped with a deep copy method that creates an entirely new object, complete with new states and transitions, that is identical to the current graph. This copy method, *checkpoint*, is called prior to each change made on the graph.

Once the user has finished constructing their graph, they can begin the process of converting the transition graph into a regular expression.

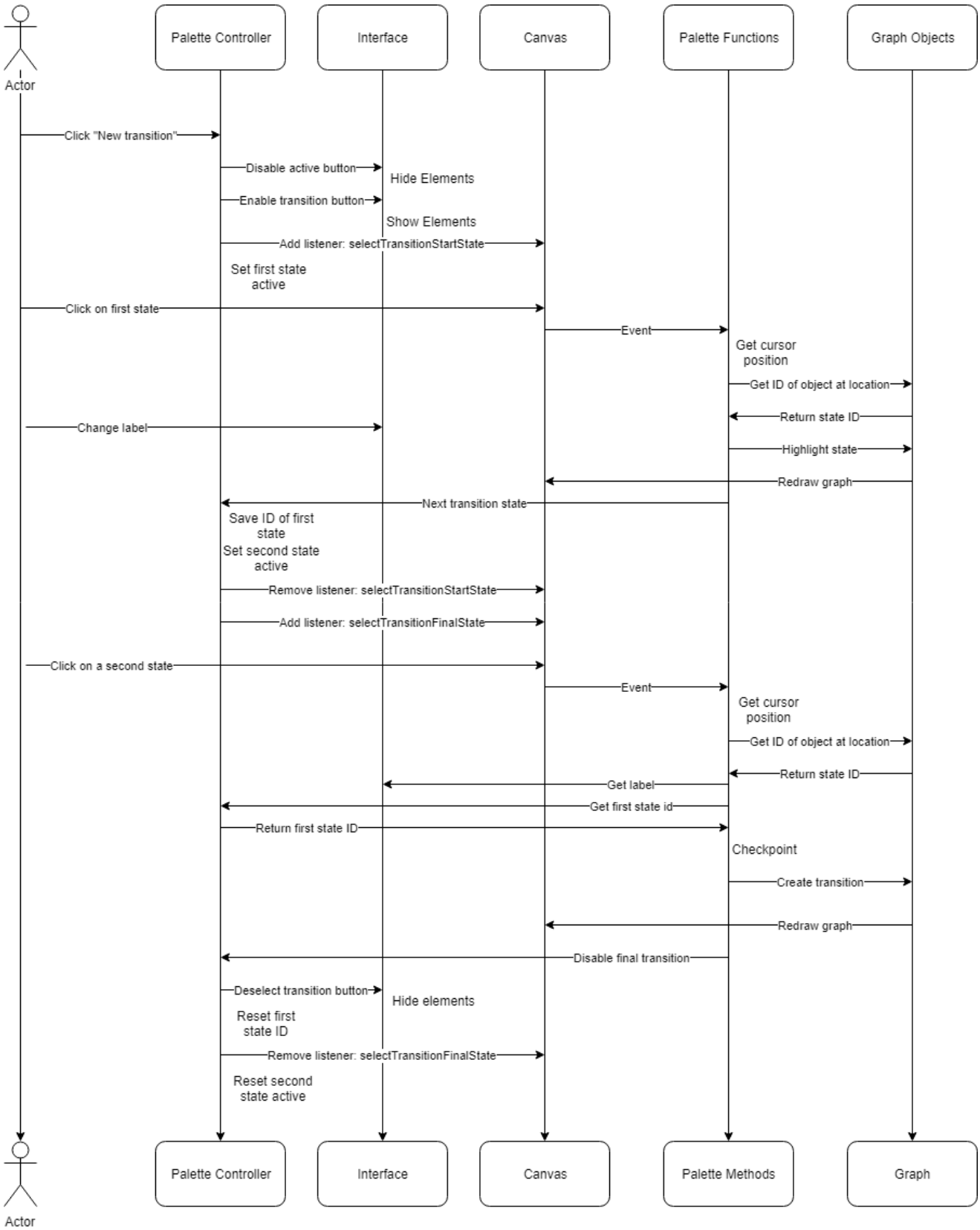

Figure 3: An activity diagram of the process of creating a new transition between two states in the graph

Verification and Conversion

String Verification

The conversion to a regular expression is a process that begins with ensuring that the created graph is a well-formed transition graph. The only state requirements to be verified are that the graph contain at least one start state. After that, each transition label must be examined to ensure that it is a well-formed regular expression composed with characters from the defined alphabet. This is not a simple task: since a regular expression can contain any number of nested parentheses, the language of all regular expressions cannot be represented with a regular expression.

After some research, I learned of a context-free grammar to solve this problem [2], paraphrased below:

RegEx	→	Concat | RegEx + Concat
Concat	→	Brack | Concat . Brack
Brack	→	Simple | Brack * | (RegEx)
Simple	→	{All terminals in the alphabet} | λ

To accommodate this grammar, the labels must first be converted to a format that can be accepted by it. After stripping the whitespace from the labels, all instances of concatenation (typically written without operators) are identified with the "." operator. Each rule in the context-free grammar is represented with its own function that accepts a string and returns true or false. The RegEx and Concat functions explore every possible separation of the input string into its two components for their second rule in order to find a match.

If all transition labels are successfully verified in this manner, the user may begin the conversion of the transition graph to a regular expression.

Regular Expression Conversion

The algorithm used for constructing regular expressions from transition graphs is as follows:

1. Create a unique start state, connecting it to all previous start states using lambda transitions.
2. Create a unique final state, connecting it to all previous final states using lambda transitions.
3. Combine all pairs of edges with the same source and destination into one edge, with a label equal to the union of the two previous edge labels (a + b).
4. Remove all single-state loops with label A, adding A* to all incoming edges of that state. If the state has no incoming transitions, instead prepend A* to the label of all its outgoing transitions.
5. Choose a state A and remove it, updating edges accordingly: For each state P that precedes A and for each state F that follows A, add the transition P→F with a label equal to the label of P→A concatenated with the label of A→F.

6. Repeat steps 3 through 5 until only two states and at most one transition remain. This final transition label, if it exists, is the regular expression. If there is no transition, the transition graph does not accept any strings.

In the interest of acting as a teaching tool, the conversion process is not automated and must be manually executed by the user. In this way, the user sees the impact of every step in the algorithm. Steps 1 through 5 in the conversion process are executed by the user individually in any order they choose.

Like the palette, these actions are monitored by a Conversion Controller object that checks the state of each action as it is performed (or cancelled) and assigns and removes listeners to the canvas as needed.

Reflection

Process

Work on the project began with a project proposal. The proposal outlined all the things I wished to accomplish with the project and was essentially a rewording of the tasks set by my supervisor. Outlining the tasks in my own words framed the project in terms of its principal goals and helped me begin planning on how to achieve them.

From the project proposal I set out to write a project description that would outline all the functionality necessary to achieve the goals set by the proposal. Within the project description I described the features of the canvas and planned out the core functionality of the Palette, as well as features to save and load graphs from XML files, export them to PNG images, and of course to verify their validity and convert them to regular expressions.

Even at this point in project development I had set aside the possibility of completing the third phase of the project. I realized that my combined workload over the course of the semester would not realistically allow for this task to be completed.

After I had completed the project description, I began work on describing the implementation of the features described. I swiftly realized that my Javascript knowledge had deteriorated somewhat since I had last used it in the summer of 2017, and that I could not possibly describe any meaningful implementation without understanding the structure of the language. Instead, I abandoned planning and jumped right into programming.

Challenges

Three challenges stood out from my work on this project. First were the calculations required to draw a loop transition on a state: a circle centered on the absolute north end of the state, sweeping from the northeast to the northwest edges. After some pondering and some unsuccessful attempts, I managed to apply an arccosine calculation to get the angle required to draw the loop exactly to meet the edges of the state, thus making good use of my high-school trigonometry lessons.

A second significant challenge I faced was in the process of verifying the user's transition graph prior to beginning the conversion to a regular expression. Converting a context-free

grammar into a series of functions was a task that resulted in several bugs that required some deep thinking to solve. Though this was an enjoyable task that broke from the relatively straightforward activity of managing the document object model, it was a task that took a large amount of effort considering the perceived size of the feature that it enabled.

Another task that consumed a significant amount of development time was the planned feature for saving and loading graphs in XML format. Converting the graph objects into XML strings was a simple task, but it was not even half the challenge of this issue. Because I was working with a local Javascript application that did not make use of any external packages or frameworks, I had great difficulty in determining a method to save and load files from the user's machine. After a few unsuccessful attempts at implementing small external packages, I abandoned the feature entirely. The methods to convert graph objects to XML strings still exist in the application, but they are inaccessible. It was disheartening to see so much development time consumed by a feature that did not see the light of day, and I felt painfully aware of my lack of aptitude for web programming that I would put so much effort into doing a task the wrong way.

Self-Evaluation

My lack of experience with web development led to a lot of sub-standard approaches to problems. Many feature implementations could have been done in more efficient and/or less complicated ways.

An example is the function that combines transitions with the same source and destination, which contains three nested loops. This complexity could potentially have been avoided had I used some form of data structure from an external library that updated dynamically or allowed for improved iteration.

As my knowledge of Javascript improved over the course of my work on the project, so did the structure and quality of the code I wrote. This resulted in some inconsistencies in program structure, especially apparent in the differences between the *PaletteController* and *ConversionController* classes. Both classes perform similar tasks – the management of the state of operations on the canvas through adding and removing event listeners and storing Boolean values. However, the complexity of the *PaletteController* class, written earlier in the development time, is much higher than that of the *ConversionController* class. The complexity of *PaletteController* is partially responsible for the convoluted process of creating a transition as described earlier in Fig. 3.

Suggested Next Steps

Because of time limitations, I was not able to implement every feature I had envisioned. There are a few areas that I believe could be improved should somebody perform more work on this project.

Chief among these areas are some key issues with the representation of the graph on the canvas. In the application's current state, multiple transitions between the same states will be rendered in the same space, with no way to tell them apart. Additionally, loop transitions do not

have an arrow at the end like regular transitions do. Fixing these issues would require significant work on the method that draws transitions, a method that is already complex as it is, and so the issues remain.

Some other potential next steps lie in the realm of user experience. Adding keyboard shortcuts to the palette commands would greatly improve the efficiency of graph construction, as would adding the ability to relocate states through clicking and dragging, thereby eliminating the need to delete and recreate them. The user interface itself is currently quite spartan and could use many improvements to the CSS styling to make the application more pleasing to the eye.

Saving and loading via XML is a feature that should be simple to implement as half the work is already completed. This task can be completed in one of two ways: one possibility is taking XML input directly via text pasted into an input element, and similarly output as text. This would require the user to save the text to a file on their own, which makes it the least preferable of the two options in terms of user experience. Alternatively, the application may save the XML data to a file server-side and provide this file to be downloaded by the client. I believe that this is a task that is easier to achieve compared to the fully client-side implementation of file creation and reading, as I attempted.

The educational aspect of the application could be improved by including more information on the material presented, whether through the form of explanations on the page or links to content elsewhere. Should the application be hosted from the course website, links might be provided to relevant lecture slides where concepts are explained.

As a single individual, there was a limit to the testing that I could perform on the application. Though the application is relatively simple, I am certain that should the application be put to use, many uncaught errors would appear.

Once all these other issues have been addressed, progress might begin on phase 3 of the application as described in the Project Goals section of the report. My approach to this would likely involve an entirely new HTML document that made use of the graph construction features of this application. New methods would be written to verify the validity of the two finite automata the user creates, and of course methods to combine them into another. My conception of this feature involves a parallel display of the regular expressions corresponding to these automata to properly demonstrate the equivalence between these two structures.

Conclusion

Given that this project was my first attempt at designing a web application from start to finish, the project was slightly out of my depth. I made many mistakes, but I learned a lot in the process. Despite all the potential areas for improvement, I am pleased to note that the finished project achieves all the goals laid out for it from the beginning, and I believe that I have left this project in a state where it can easily be improved. I am very glad to have been able to use the skills that I acquired during my undergraduate degree to contribute to the teaching of future students. I can only hope that my small tool might help those students achieve the same sense of wonder and excitement that I felt when learning about these structures myself.

References

[1] M. Lawson, Finite automata. Boca Raton: Chapman & Hall/CRC, 2004, pp. 97-113.

[2] G. Bochmann, "Introduction to context-free grammars by example : defining the language of regular expressions", Site.uottawa.ca. [Online]. Available: http://www.site.uottawa.ca/~bochmann/SEG-2106-2506/Notes/M2-3-SyntaxAnalysis/grammar-for-regular-expressions.html. [Accessed: 17- Dec- 2018].