

DELHI TECHNOLOGICAL UNIVERSITY

Subtext - Context Based Communication Application

Mini Project (MC-391) Final Report

Anish Sachdeva

DTU/MC/2K16/013





Subtext - Context Based Communication Application

Final Project Report (MC-391)

1st May 2020

Anish Sachdeva

Delhi Technological University

Under the Guidance of Prof. Dr. Sivaprasad Kumar

Acknowledgements

I would like to express my special thanks of gratitude to my teacher Prof Dr. Sivaprasad Kumar of the Mathematics department at Delhi Technological University who gave me a golden opportunity to do this wonderful project on creating a Context based communication web app, which has led me to research heavily and implement web apps that are completely RESTful and use modern technologies like MongoDB, Angular, Express, Node.js etc. and are managed in a modern way using git and maintained on Github.

I have also learnt extensively about markdown and have implemented a Lexical Parser by myself which has further led me to explore and learn about Finite Automata, Regular expressions and languages.

I have also learnt a little about Latex and how it is implemented by the LaTeX compiler generated by Donald Knuth and how we can use this prebuilt compiler to generate *dvi* files (*De vice Independent*) from utf-8 text and then generate symbols from the generated *dvi* files.

Secondly I would also like to thank my parents and friends who helped me a lot in finalizing this project within the limited time frame.

Undertaking

I Anish Sachdeva, enrolled in Delhi Technological University in 2016 in Mathematics and Computing Engineering. I took a one year gap after my 4th semester and am currently in my 6th semester. I took the elective course Mini project (MC-391) in my 5th semester under the guidance of Prof. Dr. Sivaprasad Kumar.

The project that was mutually finalized between me and my supervisor Dr. Sivaprasad Kumar was a chatting application that users can use to communicate in not just normal text messages, but also LaTeX and markdown based perfomatted messages.

This project was finalized and approved by my department Head Prof. Dr. Sangita Kansal ma'am.

I hereby declare that the mini-project I have created; the application subtext and the Markdown and LaTeX Parser used in this project are solely my work and have been built from the ground up by me.

I also declare that no help was taken from any other person or faculty and my supervisor except Prof. Dr. Sivaprasad Kumar who provided a vision and short term goals to accomplish this project.

I have used existing frameworks and databases, but that is only for the creation of the application.

Following are the projects that I have made and I deploy them as open source projects protected under the MIT license on GitHub.

1. [Subtext](#): A context based communication app deployed [here](#). This is the main running application wheras the other 2 projects are simply a library of functions that on their own do not showcase much but when imported and properly added into a project (such as SubText) showcase their true functionality.

SubText acts as a display/demo for the 2 parsers and function libraries written below.

2. [md-to-html-parser](#): A TypeScript based markdown parser that converts valid markdown into html code
3. [latex-to-html-parser](#): A TypeScript based LaTeX to html parser that converts valid LaTeX code into html





Index

Introduction	3
Motivation	4
Facebook Whatsapp	4
Facebook Messenger	5
Slack	6
Aim	7
Vision	9
Markdown	10
Finite Automata	11
Deterministic Finite State Automata (DFA)	12
Non Deterministic Finite State Automata (NFA)	13
Context Free Grammars	14
Formal Definition	15
Lexical Parsing	16
Token	16
Keywords	16
Identifiers	16
Operators	16
Separators	17
Non-Tokens	17
Lexeme	17
Functionality	17
Lexical Grammar	18
Tokenization	19
Scanner	20
Evaluator	21
Obstacles	22
Phrase Structure	22
Line Continuation	23
Semicolon Insertion	23
Off-Side Rule	23

Context-Sensitive Lexing	24
Parse Tree	25
Abstract Syntax Tree	26
LaTeX	27
LaTeX Implementation in The Project	28
Running the Project on the browser	29
Running the Project on Your Local Machine	30
Technical Specification	31
Subtext vs Overleaf LaTeX computation	32
Examples	34
Login Page	34
LaTeX Support	36
Markdown Support	38
Future Scope	40
For Casual Consumers	40
For Organizations (Future Growth Ideas for product - Optimistic Venture)	40
For Chatting inside a particular channel	41
For chatting with another user in the group	41
Markdown Parser Code Snippets	42
to-heading.ts	42
to-italics-and-bold.ts	43
to-strikethrough.ts	45
to-paragraph.ts	45
to-table.ts	47
to-code-block.ts	53
to-block-quotes.ts	54
to-emoji.ts	55
emojis.ts	55
to-list.ts	57
LaTeX Parser Code Snippets	63
sqrt.js	63
underline.js	67
arrow.js	69
math.js	73

buildHTML.js	75
parseTree.js	87
SubText Application Code	89
app.module.ts	89
app-routing.module.ts	90
chat.service.ts	91
clipboard.service.ts	93
group.service.ts	94
markdown-parser.service.ts	94
user.service.ts	95
login.component.html	97
login.component.css	97
login.component.ts	99
dashboard.component.html	100
dashboard.component.css	102
dashboard.component.ts	106
message.component.html	108
message.component.css	108
message.component.ts	109
message.ts	110
message-type.enum.ts	111
Conclusion	112
Bibliography	113

Introduction

Subtext is a real time communication web application that has been built over a period of 2 semesters i.e. 1 year to incorporate certain functionality and features that are not found in common chatting applications or communication applications these days.

It incorporates normal messaging facilities found on most applications such as peer-to-peer messages and having groups where multiple people can send messages, but the highlight of this new application are the text parsing capabilities that it offers.

SubText has the ability to parse any markdown text that the user enters in the message bar and show the user the parsed and interpolated text message in real time with no delay. The user can use this functionality to write down and frame this message as she sees fit and send when the message has been composed to her satisfaction with correct markdown transpilation.

The application also provides the functionality of transpiling to LaTeX as the user is typing. The user sees the transpiled just as she saw with Angular and just as she could send a message on the fly as she did with markdown containing LaTeX content.

The transpilation to markdown and LaTeX in a common text container that the user can use even for sending normal messages with just emojis etc are the clear advantageous user features of this application.

Furthermore the application also comes with a standard login screen where the user can enter a unique username and enter into the chatting application.

In creating this application I have learnt many new concepts based on Lexical Analysis, Lexical Parsing and Parse Trees in general and all of these concepts are based heavily on what I have learnt this semester and what I learnt in my previous semesters. So, this project is in many ways a practical application of many of the core subjects in Mathematics and Computing Engineering - MC.

The subjects that have been used heavily are Theory of Computation (MC-304), Algorithm Design and Analysis (CS-262). Software Development and Design of Web Based Applications.

Motivation

Everyone uses multiple communication apps like Whatsapp and Facebook's Messenger or even skype or slack or Google hangouts everyday and these communication apps have become integral in our workflow and getting things done. All these applications are provided by different vendors and are certainly unique in some manner or the other. We can quantify them and the features they possess.

Facebook Whatsapp

WhatsApp Messenger, or simply **WhatsApp**, is an American freeware, cross-platform messaging and Voice over IP (VoIP) service owned by Facebook, Inc. It allows users to send text messages and voice messages, make voice and video calls, and share images, documents, user locations, and other media. WhatsApp's client application runs on mobile devices but is also accessible from desktop computers, as long as the user's mobile device remains connected to the Internet while they use the desktop app. The service requires users to provide a standard cellular mobile number for registering with the service. In January 2018, WhatsApp released a standalone business app targeted at small business owners, called WhatsApp Business, to allow companies to communicate with customers who use the standard WhatsApp client.

The client application was created by WhatsApp Inc. of Mountain View, California, which was acquired by Facebook in February 2014 for approximately US\$19.3 billion. It became the world's most popular messaging application by 2015, and has over 2 billion users worldwide as of February 2020. It has become the primary means of communication in multiple countries and locations, including Latin America, the Indian subcontinent, and large parts of Europe and Africa.

The main features of this application are or the different types of contexts it can send amongst it's users are:

1. Utf8 Text message sharing
2. Link sharing
3. GIF sharing
4. Emoji (Standard ASCII) sharing
5. Contact sharing
6. Location
7. Live Location
8. Files Sharing
9. Photos Sharing

10. Recording Sharing

But the files or photos that we share are removed automatically in 30 days from the server and also if we delete them from our storage and the same goes for messages. There is no centralized message storage, but only stored on user's device and this makes whatsapp a very temporal storage medium and something that the SubText Application must address.

Facebook Messenger

Facebook Messenger (commonly known as **Messenger**) is an American messaging app and platform developed by Facebook, Inc. Originally developed as Facebook Chat in 2008, the company revamped its messaging service in 2010, and subsequently released standalone iOS and Android apps in August 2011 and standalone Facebook Portal hardware for Messenger-based calling in Q4 2018. Later on, Facebook has launched a dedicated website interface (Messenger.com), and separated the messaging functionality from the main Facebook app, allowing users to use the web interface or download one of the standalone apps. In April 2020, Facebook officially released Messenger for Desktop, which is supported on Windows 10 and macOS and distributed on Microsoft Store and App Store respectively.

Users can send messages and exchange photos, videos, stickers, audio, and files, as well as react to other users' messages and interact with bots. The service also supports voice and video calling. The standalone apps support using multiple accounts, conversations with optional end-to-end encryption, and playing games.

The contexts we can share in facebook's messenger with users are:

1. Utf8 Text message sharing
2. Link sharing
3. GIF sharing
4. Emoji (Standard ASCII) sharing
5. Location
6. Live Location
7. Files Sharing
8. Photos Sharing
9. Recording Sharing

Facebook's Messenger has similar functionality when compared with whatsapp but stores all data persistently on the internet in a server and hence can be retrieved from any device which makes the user experience much better.

We can similarly compare other applications, and they all will have a similar contextual sharing cardinality and range except for Slack.

Slack

Slack began as an internal tool for Stewart Butterfield's company Tiny Speck during the development of *Glitch*, an online game. Slack launched in August 2013.

In March 2015, Slack announced it had been hacked over four days in February 2015, and that some user data was compromised. The data included email addresses, usernames, hashed passwords, and in some cases, phone numbers and Skype IDs users had associated with their accounts. Slack added two-factor authentication to their service in response to the attacks.

Slack was previously compatible with non-proprietary Internet Relay Chat (IRC) and XMPP messaging protocols, but the company closed the corresponding gateways in May 2018.

"Searchable Log of All Conversation and Knowledge" is an acronym for "Slack".

Slack went public without an IPO on April 26, 2019 and saw its shares soar to \$21 billion valuation.

Slack is the only application that I have used or heard among the plethora of communication apps that provides markdown support for programmers and developers. Other applications that provide markdown support are also exclusively developer focussed.

The Motivation to build another communication application and add to the already long list was to create one single application that was extremely light weight and didn't feel convoluted or complex or something that only developers and programmers will use.

Rather it should feel like Whatsapp and something that everyone, irrespective of profession or technical background can use and be productive with.

Aim

The aim of this mini-project is to create a web based application called SubText that can provide the user with many ways of sharing data in different contextual forms and not just plain old utf-8 based text.

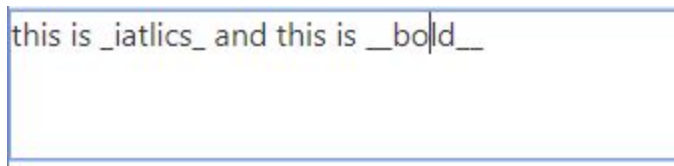
Normally, if someone is in some cooking page group on whatsapp to share some recipe they will have to share text rather than sending in a list. If someone is in the class group, then to ask the teacher a question or to ask your friends something you will need to write it using pen and paper and then click a photograph and then send that photograph. Similarly the teacher or your friend who you have asked the question will also have to write it via pen and paper to send you a picture of your solution.

Can people part of music groups share music sheets or write musical notes in staff notation which is something that they actually understand or must they also resort to taking pictures and sending pictures and replying back with pictures.

This sharing of text and photos as the only available option to exchange any and all contexts is something that I wished to change once and for all. The new application that I wanted to create - SubText should grow into a full fledged application that can one day be sold to enterprises just like Slack and also to common people like Whatsapp and provide them with an alternative to current applications.

The contexts that have been added to this application in this 2 semester period are the toughest and most gruelling implementations that have been added. They are:

1. Simple plain utf-8 text support.
2. Markdown support. The user can enter a valid markdown and see it transpiled instantaneously and also send this formatted markdown rather than copy pasted code.



```
this is _iaticls_ and this is __bold__
```



this is *iaticls* and this is **bold**

3. LaTeX Support: The persons should be able to share amongst themselves valid latex parsed text and share mathematical formulas or equations etc.

this is \LaTeX and $\sum_{i=1}^N n$

this is \LaTeX and $\sum_{i=1}^N n$

Furthermore in the future we can expand this application to understand other contexts as well such as :

1. List Sharing
2. Link Sharing and automatic link understanding such as YouTube links or Facebook links
3. Website Integrations such as other social media Integrations like Instagram link sharing, quora articles sharing, medium pages and stories sharing.
4. Staff Notation sharing for musically oriented chats.
5. Adding Handwriting recognition for staff notation and musical notes sharing.
6. Adding handwriting recognition for converting hand strokes into mathematical equations and translating handwriting directly into LaTeX which is something that can be understood, copied and shared in the digital world.
7. Adding the sharing and creating of quizzes and forms to collect data.
8. Option to create and share surveys amongst users and collect data directly within the application.

Vision

The vision of the application - SubText is to provide users with a common communication app for all their needs that is not technically demanding to learn or to use, but rather anyone who wishes to use just a plain sharing app can use from the get go without realising that multiple other contexts are even supported.

And for people that do need to share certain information or data that can be better expressed in certain forms rather than just text or images then they should have the option to do that and all user data must be stored on the server to make this application persistent (unlike Whatsapp).

The application to enhance user productivity and help them accomplish tasks with speed and accuracy.



Markdown

Markdown is a lightweight markup language with plain text formatting syntax. Its design allows it to be converted to many output formats, but the original tool by the same name only supports HTML. Markdown is often used to format readme files, for writing messages in online discussion forums, and to create rich text using a plain text editor.

Since the initial description of Markdown contained ambiguities and unanswered questions, the implementations that appeared over the years have subtle differences and many come with syntax extensions.

John Gruber created the Markdown language in 2004 in collaboration with Aaron Swartz on the syntax, with the goal of enabling people "to write using an easy-to-read and easy-to-write plain text format, optionally convert it to structurally valid XHTML (or HTML)".

Its key design goal is *readability* – that the language be readable as-is, without looking like it has been marked up with tags or formatting instructions, unlike text formatted with a markup language, such as Rich Text Format (RTF) or HTML, which have obvious tags and formatting instructions. To this end, its main inspiration is the existing conventions for marking up plain text in email, though it also draws from earlier markup languages, notably setext, Textile, and reStructuredText.

Gruber wrote a Perl script, `Markdown.pl`, which converts marked-up text input to valid, well-formed XHTML or HTML and replaces angle brackets '`<`' '`>`' and ampersands '`&`' with their corresponding character entity references. It can be used as a standalone script, as a plugin for Bloxom or Movable Type, or as a text filter for BBEdit.

Markdown has since been re-implemented by others e.g. in a Perl module available on CPAN (`Text::Markdown`), and in a variety of other programming languages. It is distributed under a BSD-style license and is included with, or available as a plugin for, several content-management systems.

Sites like GitHub, Bitbucket, Reddit, Diaspora, Stack Exchange, OpenStreetMap, and SourceForge use variants of Markdown to facilitate discussion between users.

Finite Automata

A Finite Automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the **states**
2. Σ is a finite set called the **alphabet**
3. $\delta : Q \times \Sigma \rightarrow Q$ is the **transition function**
4. $q_0 \in Q$ is the **start state**, and
5. $F \subseteq Q$ is the **set of accepted states**.

A finite automata can also be represented by its corresponding state transition diagram.

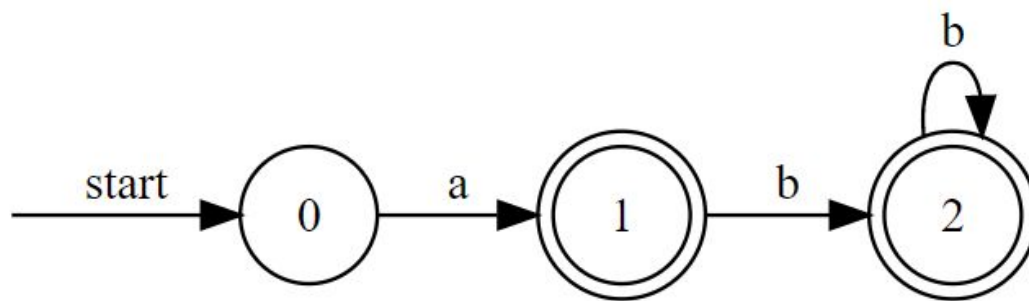


Figure 1: Finite Automata that recognizes all strings that start with exactly 1 'a' and may contain an arbitrary number of 'b' after the 'a'.

This finite state automata can also be represented with a state transition table that indicates which state it will go to after encountering a particular symbol. Φ here denotes a null state where once the automata has entered can never come out irrespective of the symbol it encountered. It is an absorbing state.

State	a	b
→ 0	1	Φ
1	Φ	2
2	Φ	2

Figure 2: The state transition table for the finite automata represented in Figure 1

Deterministic Finite State Automata (DFA)

A deterministic finite state automata is a finite automata where given a state and an encountered symbol the machine will go into only one new (or same) state and the next state can be uniquely *determined* given the symbol and current state.

This isn't true for non-deterministic machines. Some examples of finite state automata are as follows:

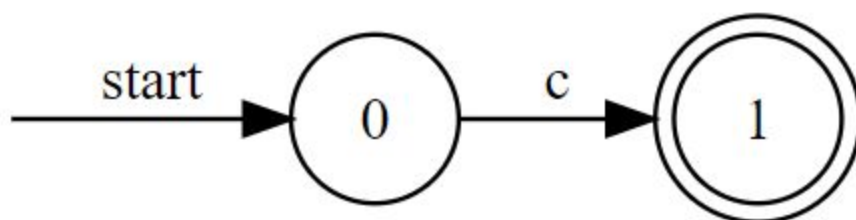


Figure 3: Deterministic Finite State Automata that recognizes only the symbol 'c'

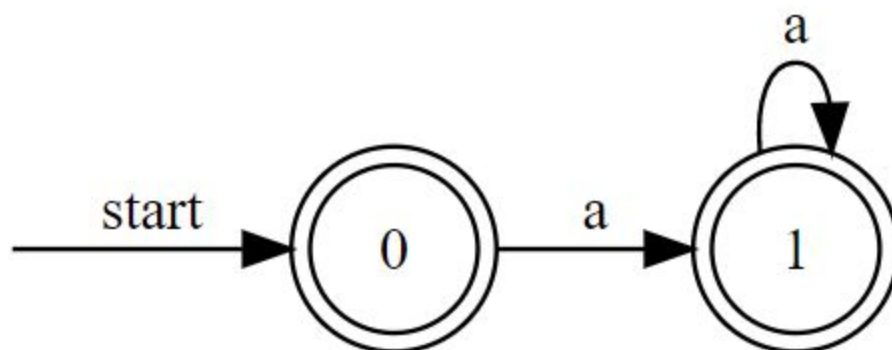


Figure 4: Deterministic Finite State Automata that recognizes the null string ϵ or any number of 'a'

Non Deterministic Finite State Automata (NFA)

A Finite Automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the **states**
2. Σ is a finite set called the **alphabet**
3. $\delta : Q \times \Sigma \rightarrow P(Q)$ is the **transition function**
4. $q_0 \in Q$ is the **start state**, and
5. $F \subseteq Q$ is the **set of accepted states**.

Where $P(Q)$ is the power set of the states. A non-deterministic finite state automata can also be represented by its corresponding state transition diagram.

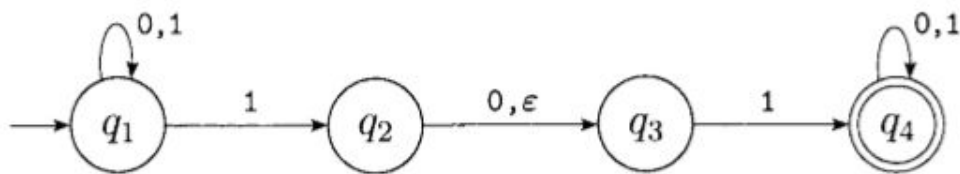


Figure 4: Non Deterministic Finite State Automata that recognizes all strings that contain the substrings as '11' or '101'

The formal definition of this automata is $(Q, \Sigma, \delta, q_0, F)$, where

1. $Q = \{q_1, q_2, q_3, q_4\}$
2. $\Sigma = \{0, 1\}$
3. δ is given as

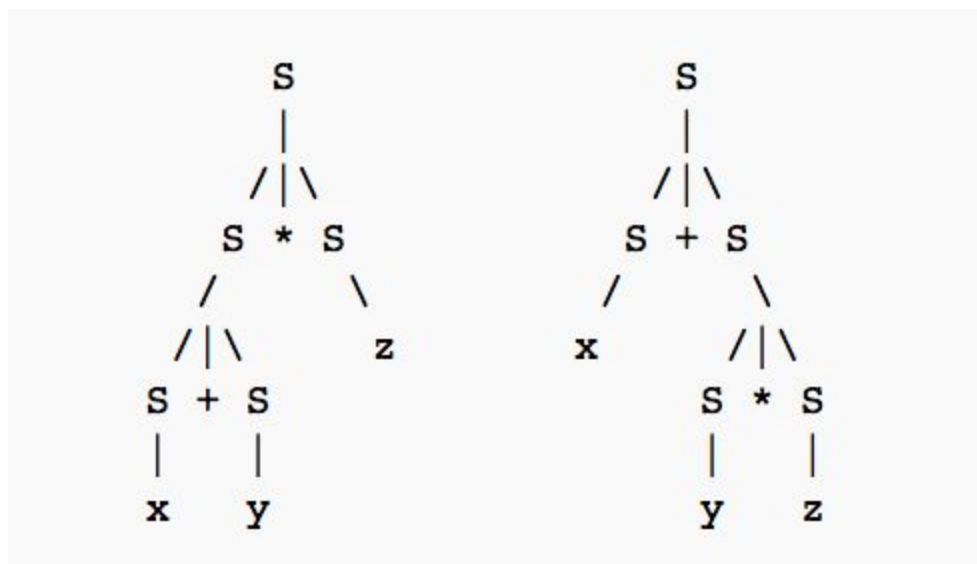
	0	1	ϵ
q_1	$\{q_1\}$	$\{q_1, q_2\}$	\emptyset
q_2	$\{q_3\}$	\emptyset	$\{q_3\}$
q_3	\emptyset	$\{q_4\}$	\emptyset
q_4	$\{q_4\}$	$\{q_4\}$	\emptyset

4. q_1 is the start state
5. $F = \{q_4\}$

Context Free Grammars

Context-free grammars (CFGs) are used to describe context-free languages. A context-free grammar is a set of recursive rules used to generate patterns of strings. A context-free grammar can describe all regular languages and more, but they cannot describe *all* possible languages.

Context-free grammars are studied in fields of theoretical computer science, compiler design, and linguistics. CFG's are used to describe programming languages and parser programs in compilers can be generated automatically from context-free grammars.



Context-free grammars can generate context-free languages. They do this by taking a set of variables which are defined recursively, in terms of one another, by a set of production rules. Context-free grammars are named as such because any of the production rules in the grammar can be applied regardless of context—it does not depend on any other symbols that may or may not be around a given symbol that is having a rule applied to it.

Context-free grammars have the following components:

- A set of terminal symbols which are the characters that appear in the language/strings generated by the grammar. Terminal symbols never appear on the left-hand side of the production rule and are always on the right-hand side.

- A set of nonterminal symbols (or variables) which are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols. These are the symbols that will always appear on the left-hand side of the production rules, though they can be included on the right-hand side. The strings that a CFG produces will contain only symbols from the set of nonterminal symbols.
- A set of production rules which are the rules for replacing nonterminal symbols. Production rules have the following form: variable \rightarrow string of variables and terminals.
- A start symbol which is a special nonterminal symbol that appears in the initial string generated by the grammar.

Formal Definition

A context free grammar can be described by a 4 element tuple (V, Σ, R, S) , where

V is a finite set of variables (which are non-terminal)

Σ is a finite set (disjoint from V) of terminal symbols

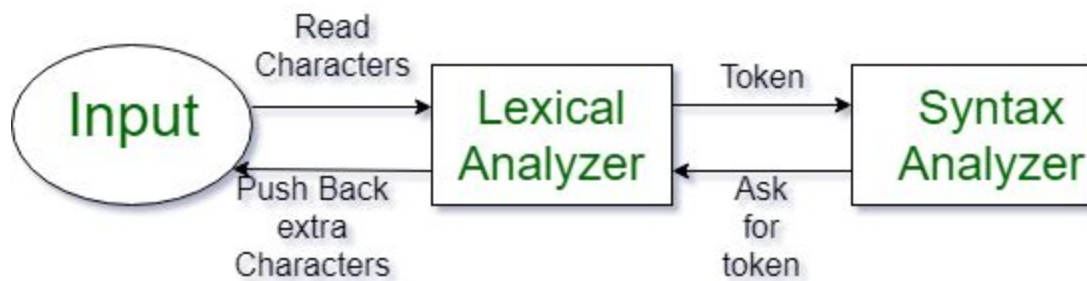
R is a set of production rules where each production rule maps a variable to a string $s \in (V \cup \Sigma)^*$

S (which is in V) which is a start symbol.

Lexical Parsing

Lexical Analysis is the first phase of compilers also known as scanners. It converts the High level input program into a sequence of **Tokens**.

- Lexical Analysis can be implemented with the Deterministic finite Automata.
- The output is a sequence of tokens that is sent to the parser for syntax analysis.



Token

A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.

Examples of Tokens:

1. Type token (id, number, real ...)
2. Punctuation Tokens (if, void, return)
3. Alphabetic Tokens (keywords)

Keywords

Examples are `for`, `while`, `if` etc.

Identifiers

Examples are variable names, function names etc.

Operators

Operators are both Unary and Binary operators that we use in our programming language like `-`, `+`, `--`, `++`, `&`, `|`, `&&`, `||` etc.

Separators

These are tokens that separate constructs of code like , and ; .

Non-Tokens

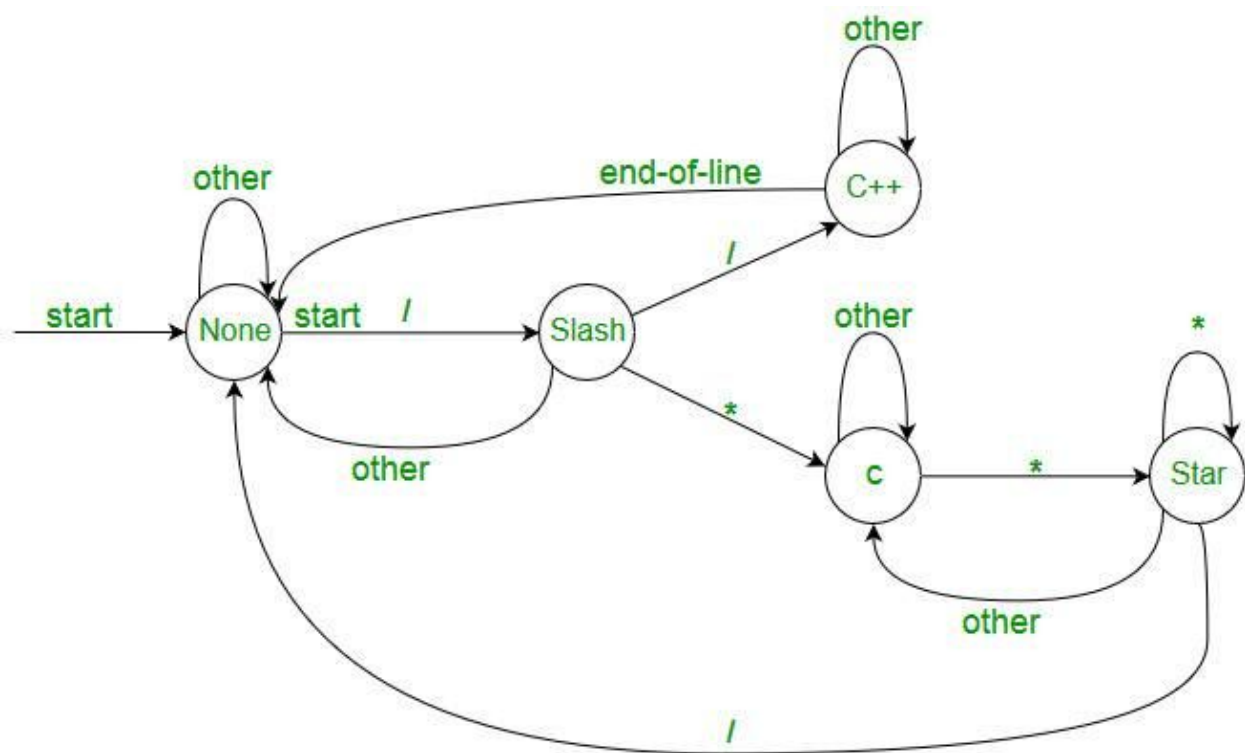
Some Non token symbols are comments, preprocessor directives, macros, blanks, tabs, new lines etc.

Lexeme

The sequence of characters matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme. eg- "float", "abs_zero_Kelvin", "=", "-", "273", ",", " .

Functionality

1. Tokenization .i.e Dividing the program into valid tokens.
2. Remove white space characters.
3. Remove comments.
4. It also provides help in generating error messages by providing row number and column number.



The lexical analyzer identifies the error with the help of an automation machine and the grammar of the given language on which it is based like C, C++ and gives row number and column number of the error.

Suppose we pass a statement through lexical analyzer

```
a = b + c
```

It will generate a token sequence as:

```
id = id + id
```

Where each id reference to its variable in the symbol table referencing all details. For example consider the program:

```
int main() {
    // 2 variables
    int a, b;
    a = 10;
    return 0;
}
```

All valid tokens are: `int main () { int a , b ; a = 10 ; return 0 ; }`

Lexical Grammar

The specification of a programming language often includes a set of rules, the lexical grammar, which defines the lexical syntax. The lexical syntax is usually a regular language, with the grammar rules consisting of regular expressions; they define the set of possible character sequences (lexemes) of a token. A lexer recognizes strings, and for each kind of string found the lexical program takes an action, most simply producing a token.

Two important common lexical categories are white space and comments. These are also defined in the grammar and processed by the lexer, but may be discarded (not producing any tokens) and considered *non-significant*, at most separating two tokens (as in `if x` instead of `if x`). There are two important exceptions to this. First, in off-side rule languages that delimit blocks with indenting, initial whitespace is significant, as it determines block structure, and is generally handled at the lexer level; see phrase structure, below. Secondly, in some uses of lexers, comments and whitespace must be preserved – for example, a

prettyprinter also needs to output the comments and some debugging tools may provide messages to the programmer showing the original source code. In the 1960s, notably for ALGOL, whitespace and comments were eliminated as part of the line reconstruction phase (the initial phase of the compiler frontend), but this separate phase has been eliminated and these are now handled by the lexer.

Tokenization

Tokenization is the process of demarcating and possibly classifying sections of a string of input characters. The resulting tokens are then passed on to some other form of processing. The process can be considered a sub-task of parsing input.

The string isn't implicitly segmented on spaces, as a natural language speaker would do. The raw input, the 43 characters, must be explicitly split into the 9 tokens with a given space delimiter (i.e., matching the string " " or regular expression `/\s{1}/`).

The tokens could be represented in XML,

```
<sentence>
  <word>The</word>
  <word>quick</word>
  <word>brown</word>
  <word>fox</word>
  <word>jumps</word>
  <word>over</word>
  <word>the</word>
  <word>lazy</word>
  <word>dog</word>
</sentence>
```

Or as an s-expression

```
(sentence
  (word The)
  (word quick)
  (word brown)
  (word fox)
  (word jumps)
  (word over)
  (word the)
  (word lazy)
  (word dog) )
```

When a token class represents more than one possible lexeme, the lexer often saves enough information to reproduce the original lexeme, so that it can be used in semantic analysis. The parser typically retrieves this information from the lexer and stores it in the

abstract syntax tree. This is necessary in order to avoid information loss in the case of numbers and identifiers.

Tokens are identified based on the specific rules of the lexer. Some methods used to identify tokens include: regular expressions, specific sequences of characters termed a flag, specific separating characters called delimiters, and explicit definition by a dictionary. Special characters, including punctuation characters, are commonly used by lexers to identify tokens because of their natural use in written and programming languages.

Tokens are often categorized by character content or by context within the data stream. Categories are defined by the rules of the lexer. Categories often involve grammar elements of the language used in the data stream. Programming languages often categorize tokens as identifiers, operators, grouping symbols, or by data type. Written languages commonly categorize tokens as nouns, verbs, adjectives, or punctuation. Categories are used for post-processing of the tokens either by the parser or by other functions in the program.

A lexical analyzer generally does nothing with combinations of tokens, a task left for a parser. For example, a typical lexical analyzer recognizes parentheses as tokens, but does nothing to ensure that each "(" is matched with a ")".

When a lexer feeds tokens to the parser, the representation used is typically an enumerated list of number representations. For example, "Identifier" is represented with 0, "Assignment operator" with 1, "Addition operator" with 2, etc.

Tokens are defined often by regular expressions, which are understood by a lexical analyzer generator such as `lex`. The lexical analyzer (generated automatically by a tool like `lex`, or hand-crafted) reads in a stream of characters, identifies the lexemes in the stream, and categorizes them into tokens. This is termed *tokenizing*. If the lexer finds an invalid token, it will report an error.

Following tokenizing is parsing. From there, the interpreted data may be loaded into data structures for general use, interpretation, or compiling.

Scanner

The first stage, the *scanner*, is usually based on a finite-state machine (FSM). It has encoded within it information on the possible sequences of characters that can be contained within any of the tokens it handles (individual instances of these character sequences are termed lexemes). For example, an *integer* lexeme may contain any sequence of numerical digit characters. In many cases, the first non-whitespace character can be used to deduce the kind of token that follows and subsequent input characters are then processed one at a time until reaching a character that is not in the set of characters acceptable for that token (this is termed the *maximal munch*, or *longest match*, rule). In some languages, the lexeme creation rules are more complex and may involve backtracking over previously read

characters. For example, in C, one 'L' character is not enough to distinguish between an identifier that begins with 'L' and a wide-character string literal.

Evaluator

A lexeme, however, is only a string of characters known to be of a certain kind (e.g., a string literal, a sequence of letters). In order to construct a token, the lexical analyzer needs a second stage, the *evaluator*, which goes over the characters of the lexeme to produce a *value*. The lexeme's type combined with its value is what properly constitutes a token, which can be given to a parser. Some tokens such as parentheses do not really have values, and so the evaluator function for these can return nothing: only the type is needed. Similarly, sometimes evaluators can suppress a lexeme entirely, concealing it from the parser, which is useful for whitespace and comments. The evaluators for identifiers are usually simple (literally representing the identifier), but may include some unstripping. The evaluators for integer literals may pass the string on (deferring evaluation to the semantic analysis phase), or may perform evaluation themselves, which can be involved for different bases or floating point numbers. For a simple quoted string literal, the evaluator needs to remove only the quotes, but the evaluator for an escaped string literal incorporates a lexer, which unescapes the escape sequences.

For example, in the source code of a computer program, the string

```
net_worth_future = (assets - liabilities);
```

might be converted into the following lexical token stream; whitespace is suppressed and special characters have no value:

```
IDENTIFIER net_worth_future  
EQUALS  
OPEN_PARENTHESIS  
IDENTIFIER assets  
MINUS  
IDENTIFIER liabilities  
CLOSE_PARENTHESIS  
SEMICOLON
```

Due to licensing restrictions of existing parsers, it may be necessary to write a lexer by hand. This is practical if the list of tokens is small, but in general, lexers are generated by automated tools. These tools generally accept regular expressions that describe the tokens allowed in the input stream. Each regular expression is associated with a production rule in the lexical grammar of the programming language that evaluates the lexemes matching the regular expression. These tools may generate source code that can be compiled and executed or construct a state transition table for a finite-state machine (which is plugged into template code for compiling and executing).

Regular expressions compactly represent patterns that the characters in lexemes might follow. For example, for an English-based language, an IDENTIFIER token might be any English alphabetic character or an underscore, followed by any number of instances of ASCII alphanumeric characters and/or underscores. This could be represented compactly by the string `[a-zA-Z_][a-zA-Z_0-9]*`. This means "any character a-z, A-Z or _, followed by 0 or more of a-z, A-Z, _ or 0-9".

Regular expressions and the finite-state machines they generate are not powerful enough to handle recursive patterns, such as "*n* opening parentheses, followed by a statement, followed by *n* closing parentheses." They are unable to keep count, and verify that *n* is the same on both sides, unless a finite set of permissible values exists for *n*. It takes a full parser to recognize such patterns in their full generality. A parser can push parentheses on a stack and then try to pop them off and see if the stack is empty at the end (see example in the *Structure and Interpretation of Computer Programs* book).

Obstacles

Typically, tokenization occurs at the word level. However, it is sometimes difficult to define what is meant by a "word". Often a tokenizer relies on simple heuristics, for example:

- Punctuation and whitespace may or may not be included in the resulting list of tokens.
- All contiguous strings of alphabetic characters are part of one token; likewise with numbers.
- Tokens are separated by whitespace characters, such as a space or line break, or by punctuation characters.


In languages that use inter-word spaces (such as most that use the Latin alphabet, and most programming languages), this approach is fairly straightforward. However, even here there are many edge cases such as contractions, hyphenated words, emoticons, and larger constructs such as URLs (which for some purposes may count as single tokens). A classic example is "New York-based", which a naive tokenizer may break at the space even though the better break is (arguably) at the hyphen.

Tokenization is particularly difficult for languages written in scriptio continua which exhibit no word boundaries such as Ancient Greek, Chinese, or Thai. Agglutinative languages, such as Korean, also make tokenization tasks complicated.

Some ways to address the more difficult problems include developing more complex heuristics, querying a table of common special-cases, or fitting the tokens to a language model that identifies collocations in a later processing step.

Phrase Structure

Lexical analysis mainly segments the input stream of characters into tokens, simply grouping the characters into pieces and categorizing them. However, the lexing may be



significantly more complex; most simply, lexers may omit tokens or insert added tokens. Omitting tokens, notably whitespace and comments, is very common, when these are not needed by the compiler. Less commonly, added tokens may be inserted. This is done mainly to group tokens into statements, or statements into blocks, to simplify the parser.

Line Continuation

Line continuation is a feature of some languages where a newline is normally a statement terminator. Most often, ending a line with a backslash (immediately followed by a newline) results in the line being *continued* – the following line is *joined* to the prior line. This is generally done in the lexer: the backslash and newline are discarded, rather than the newline being tokenized. Examples include bash, other shell scripts and Python.

Semicolon Insertion


Many languages use the semicolon as a statement terminator. Most often this is mandatory, but in some languages the semicolon is optional in many contexts. This is mainly done at the lexer level, where the lexer outputs a semicolon into the token stream, despite one not being present in the input character stream, and is termed *semicolon insertion* or *automatic semicolon insertion*. In these cases, semicolons are part of the formal phrase grammar of the language, but may not be found in input text, as they can be inserted by the lexer. Optional semicolons or other terminators or separators are also sometimes handled at the parser level, notably in the case of trailing commas or semicolons.

Semicolon insertion is a feature of BCPL and its distant descendant Go, though it is absent in B or C. Semicolon insertion is present in JavaScript, though the rules are somewhat complex and much-criticized; to avoid bugs, some recommend always using semicolons, while others use initial semicolons, termed defensive semicolons, at the start of potentially ambiguous statements.

Semicolon insertion (in languages with semicolon-terminated statements) and line continuation (in languages with newline-terminated statements) can be seen as complementary: semicolon insertion adds a token, even though newlines generally do *not* generate tokens, while line continuation prevents a token from being generated, even though newlines generally *do* generate tokens.

Off-Side Rule

The off-side rule (blocks determined by indenting) can be implemented in the lexer, as in Python, where increasing the indenting results in the lexer emitting an INDENT token, and decreasing the indenting results in the lexer emitting a DEDENT token. These tokens correspond to the opening brace `{` and closing brace `}` in languages that use braces for blocks, and means that the phrase grammar does not depend on whether braces or indenting are used. This requires that the lexer hold state, namely the current indent level,



and thus can detect changes in indenting when this changes, and thus the lexical grammar is not context-free: INDENT–DEDENT depends on the contextual information of prior indent level.

Context-Sensitive Lexing

Generally lexical grammars are context-free, or almost so, and thus require no looking back or ahead, or backtracking, which allows a simple, clean, and efficient implementation. This also allows simple one-way communication from lexer to parser, without needing any information flowing back to the lexer.

There are exceptions, however. Simple examples include: semicolon insertion in Go, which requires looking back one token; concatenation of consecutive string literals in Python, which requires holding one token in a buffer before emitting it (to see if the next token is another string literal); and the off-side rule in Python, which requires maintaining a count of indent level (indeed, a stack of each indent level). These examples all only require lexical context, and while they complicate a lexer somewhat, they are invisible to the parser and later phases.

A more complex example is the lexer hack in C, where the token class of a sequence of characters cannot be determined until the semantic analysis phase, since typedef names and variable names are lexically identical but constitute different token classes. Thus in the hack, the lexer calls the semantic analyzer (say, symbol table) and checks if the sequence requires a typedef name. In this case, information must flow back not from the parser only, but from the semantic analyzer back to the lexer, which complicates design.

Parse Tree

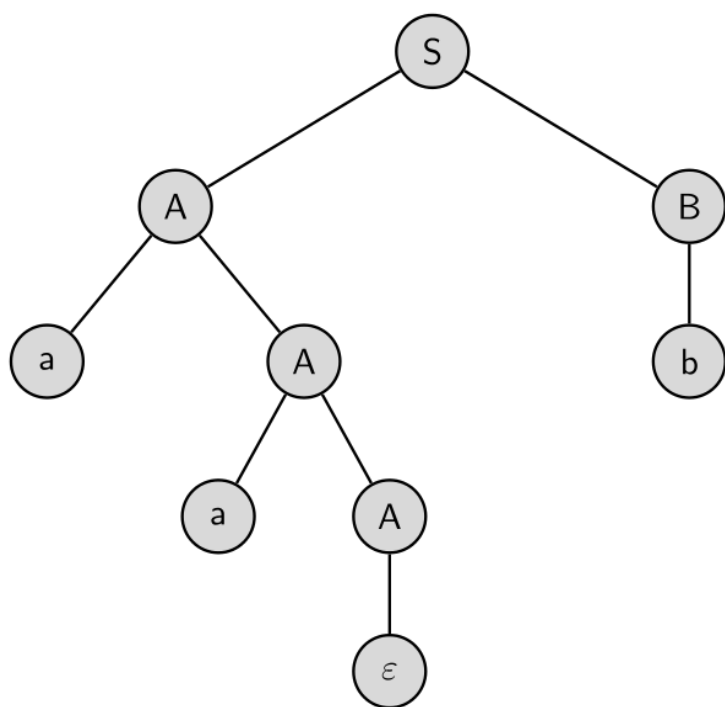
A **parse tree** or **parsing tree** or **derivation tree** or **concrete syntax tree** is an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar. The term *parse tree* itself is used primarily in computational linguistics; in theoretical syntax, the term *syntax tree* is more common.

Parse trees concretely reflect the syntax of the input language, making them distinct from the abstract syntax trees used in computer programming. Unlike Reed-Kellogg sentence diagrams used for teaching grammar, parse trees do not use distinct symbol shapes for different types of constituents.

Parse trees are usually constructed based on either the constituency relation of constituency grammars (phrase structure grammars) or the dependency relation of dependency grammars. Parse trees may be generated for sentences in natural languages, as well as during processing of computer languages, such as programming languages.

A related concept is that of **phrase marker** or **P-marker**, as used in transformational generative grammar. A phrase marker is a linguistic expression marked as to its phrase structure. This may be presented in the form of a tree, or as a bracketed expression.

Phrase markers are generated by applying phrase structure rules, and themselves are subject to further transformational rules. A set of possible parse trees for a syntactically ambiguous sentence is called a "parse forest."



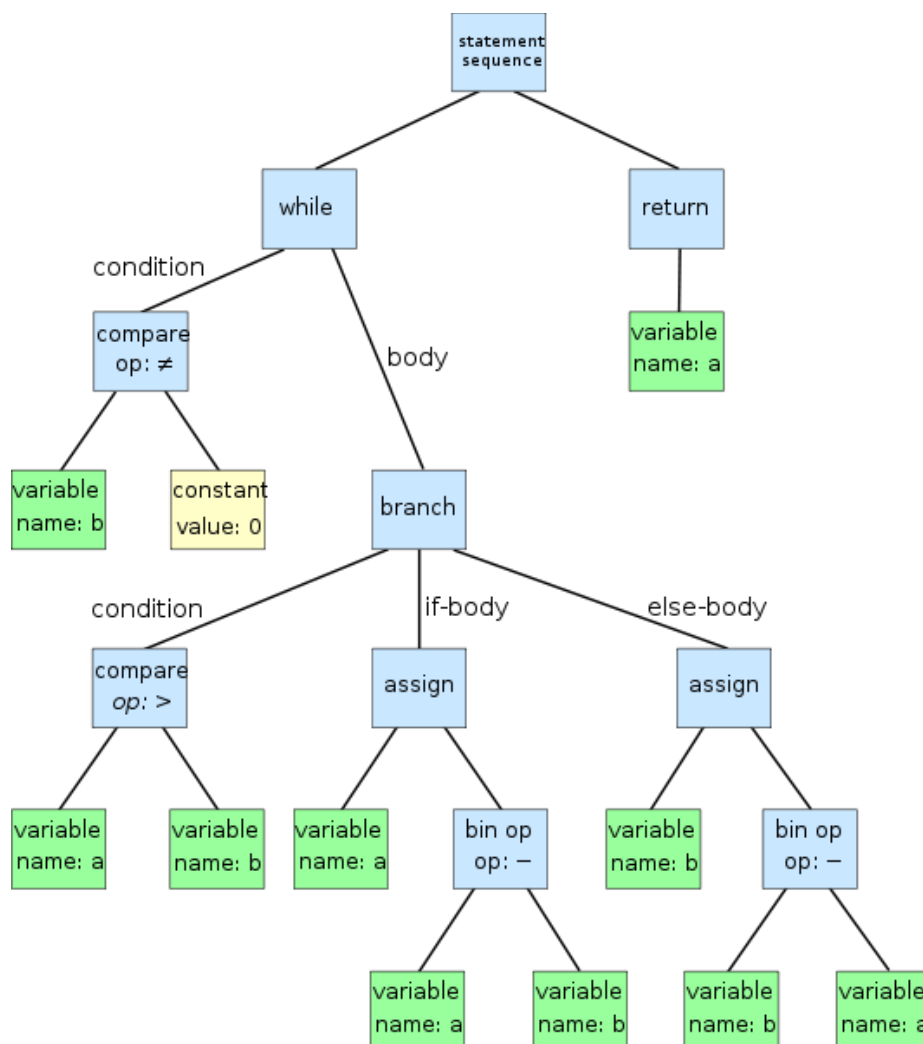
Abstract Syntax Tree

In computer science, an **abstract syntax tree (AST)**, or just **syntax tree**, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code.

The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details. For instance, grouping parentheses are implicit in the tree structure, so these do not have to be represented as separate nodes. Likewise, a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches.

This distinguishes abstract syntax trees from concrete syntax trees, traditionally designated parse trees. Parse trees are typically built by a parser during the source code translation and compiling process. Once built, additional information is added to the AST by means of subsequent processing, e.g., contextual analysis.

Abstract syntax trees are also used in program analysis and program transformation systems.



LaTeX

LaTeX is a high-quality typesetting system; it includes features designed for the production of technical and scientific documentation. LaTeX is the de facto standard for the communication and publication of scientific documents.

LaTeX is widely used in academia for the communication and publication of scientific documents in many fields, including mathematics, statistics, computer science, engineering, chemistry, physics, economics, linguistics, quantitative psychology, philosophy, and political science. It also has a prominent role in the preparation and publication of books and articles that contain complex multilingual materials, such as Sanskrit and Greek. LaTeX uses the TeX typesetting program for formatting its output, and is itself written in the TeX macro language.



LaTeX was created in the early 1980s by Leslie Lamport, when he was working at SRI. He needed to write TeX macros for his own use, and thought that with a little extra effort he could make a general package usable by others. Peter Gordon, an editor at Addison-Wesley, convinced him to write a LaTeX user's manual for publication (Lamport was initially skeptical that anyone would pay money for it); it came out in 1986¹ and sold hundreds of thousands of copies. Meanwhile, Lamport released versions of his LaTeX macros in 1984 and 1985. On 21 August 1989, at a TeX Users Group (TUG) meeting at Stanford, Lamport agreed to turn over maintenance and development of LaTeX to Frank Mittelbach. Mittelbach, along with Chris Rowley and Rainer Schöpf, formed the LaTeX3 team; in 1994, they released LaTeX 2e, the current standard version, and continue working on LaTeX3.

LaTeX Implementation in The Project

A LaTeX parser was implemented after extreme effort and painstaking amounts of code to properly parse text and convert them into LaTeX and then corresponding HTML structure that preserves both the legibility, structure and initial context that would have been generated in LaTeX.

The TeX Compiler which can compile down simple text to a tex equation was minified and embedded in the client-side code. Then the output from this compiler which is generated as a DVI file is parsed and reduced to symbols and text phrases.

This parsed output is then used to create a lexical token tree. The Lexical tree is then used to construct an HTML output by using this tree to create the same LaTeX output using HTML tags.

A special font was also imported through CSS and this and the monospace font were combined to create a union of fonts to provide space for the LaTeX typeset. The HTML output is then simply displayed in the browser which seems like LaTeX to the user of the application but just as markdown is actually just plain old HTML.

$$f(x) = \int_{-\infty}^{\infty} e^{2i\pi x} dx$$

$$f(x) = \int_{-\infty}^{\infty} e^{2i\pi x} dx$$

Running the Project on the browser

The application has been deployed on Google's Firebase hosting service and can be used simultaneously by multiple people. Google's Firebase realtime database is also being used so users can send each other messages in realtime and see changes as they do so.

The project has been deployed [here](#) and can be viewed with the given link.

When you open the application please login with some unique username (most likely your own name) and you can then send messages with the same name and retain your messages and continue your conversations even after logging out and logging in again. The application has data persistence.

Running the Project on Your Local Machine

To run this project locally you must have the following software/packages installed on your local machine:

1. [Git](#)
2. [Node](#)
3. [Angular](#)
4. [TypeScript](#)

Angular and TypeScript can be installed after installing node on your machine. To check if node and npm have been successfully installed on your machine run the following commands:

```
npm --version  
node --version
```

To add angular run the following command after adding node:

```
npm i -g @angular/angular-cli
```

To see if angular has been successfully installed run:

```
ng --version
```

To add typescript run the following command after adding node:

```
npm i -g typescript
```

Clone the repository on your machine using:

```
git clone https://www.github.com/anishLearnsToCode/subtext.git
```

To run locally:

```
cd subtext  
ng serve
```

This will now start running the web application on your localhost port 4200, which can be accessed by your web browser (prefer Google Chrome or Mozilla) at localhost:4200/.

Technical Specification

This project has been built completely as a single-page client side application on Google's Angular and for the database a realtime database using Google's Firebase has been used.

The project has been hosted and deployed on Google's Firebase and all data changes such as new messages or new users are updated directly in the database and the database being used is a real time database. Any change made in the database is propagated back to all connected applications at that instance and instantaneously propagated to all components and pages in that application.

So, this implies that someone can be using the application without ever refreshing or reloading the application. If you need to refresh the application, then you need to do nothing at all. It is guaranteed that the application is and always will be in the most latest and up-to date stage as long as your device is connected to the internet.

Now, the application consists of 3 simple groups; mathematics, Physics and Computer Science. These groups were arbitrarily created just to showcase the power of the transpiler that was created for this application.

The Computer Science group's messaging panel transpiled to Markdown whereas the Mathematics and Physics Messaging panel transpiled from LaTeX. This is just to showcase the capability and features such as creating a new group, or creating a new chat have been avoided as the main purpose of this project was to add the transpiling functionality which was the most time consuming and laborious task.

The transpiling happens on the client side and happens without any resources in an offline manner. The transpiler code is sent as a js to the client device and this code manages transpilation and also maintains the application's state by constantly being in connection with the database and on any change updates the app without needing the user to refresh at all.

Subtext vs Overleaf LaTeX computation

Overleaf is a very popular online LaTeX compilation application, but it is not a communication application but rather a full fledged online LaTeX editor where someone can create feature length documents, presentations, research papers, scientific papers etc.

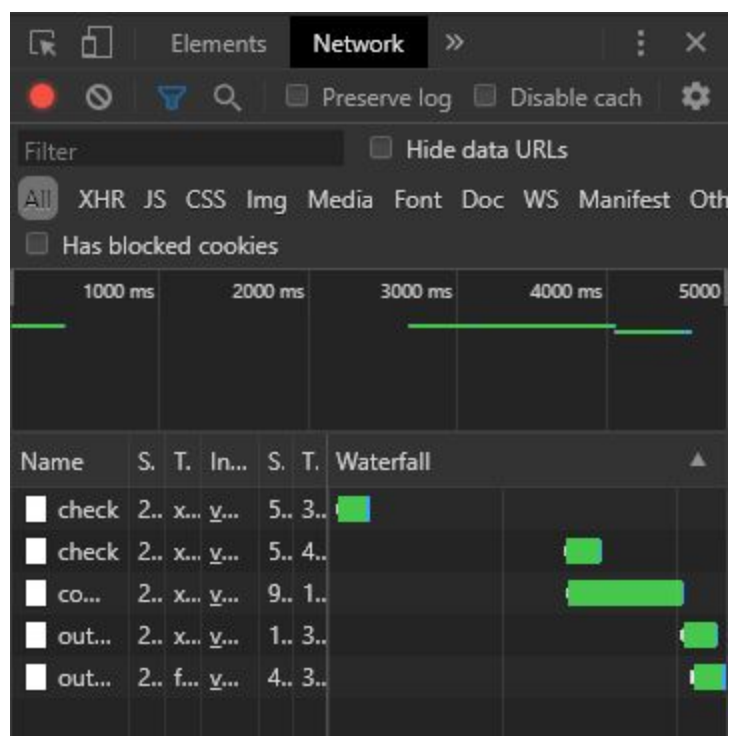
The purpose that overleaf serves and my application serves, the audience, mission, vision and aim are completely different. However they are similar on the count that both web applications are converting text written in the .tex format into LaTeX and presenting in pdf (as is the case with overleaf) or presenting it in html as is the case with my application - SubText.

Hence, it is fair to compare these 2 applications solely on the basis of their computation prowess.

I am very pleased to say that my application - SubText created by one single developer who is only in his 6th semester has outstripped a web application created by a large organization and has won the speed race by a factor of whopping 120%+.

The application SubText transpiles to LaTeX with speed and accuracy without any apparent load times and beats the Overleaf web application by a minimum factor of 120x and even 600x or 750x in many cases.

On average it is faster than Overleaf by a factor of 456x.



Here we can see that Overleaf took 5000ms for the standard test text:

```
\documentclass{article}

\usepackage[utf8]{inputenc}


\title{Test Project}
\author{anishLearnsToCode}
\date{May 2020}


\begin{document}


\maketitle


\section{Introduction}


\LaTeX
Thus is test

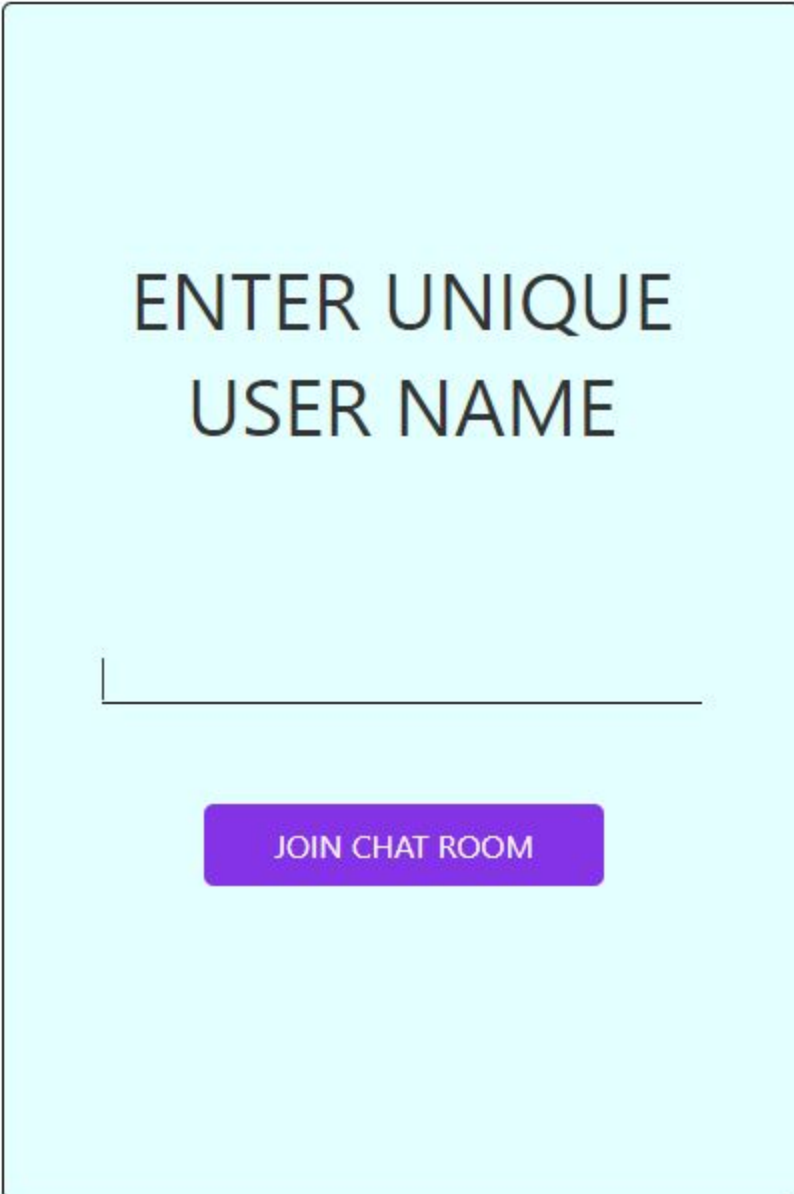

\end{document}
```

Whereas the same test text took less than 20ms in SubText which gives it an edge of 250x.

Examples

Login Page

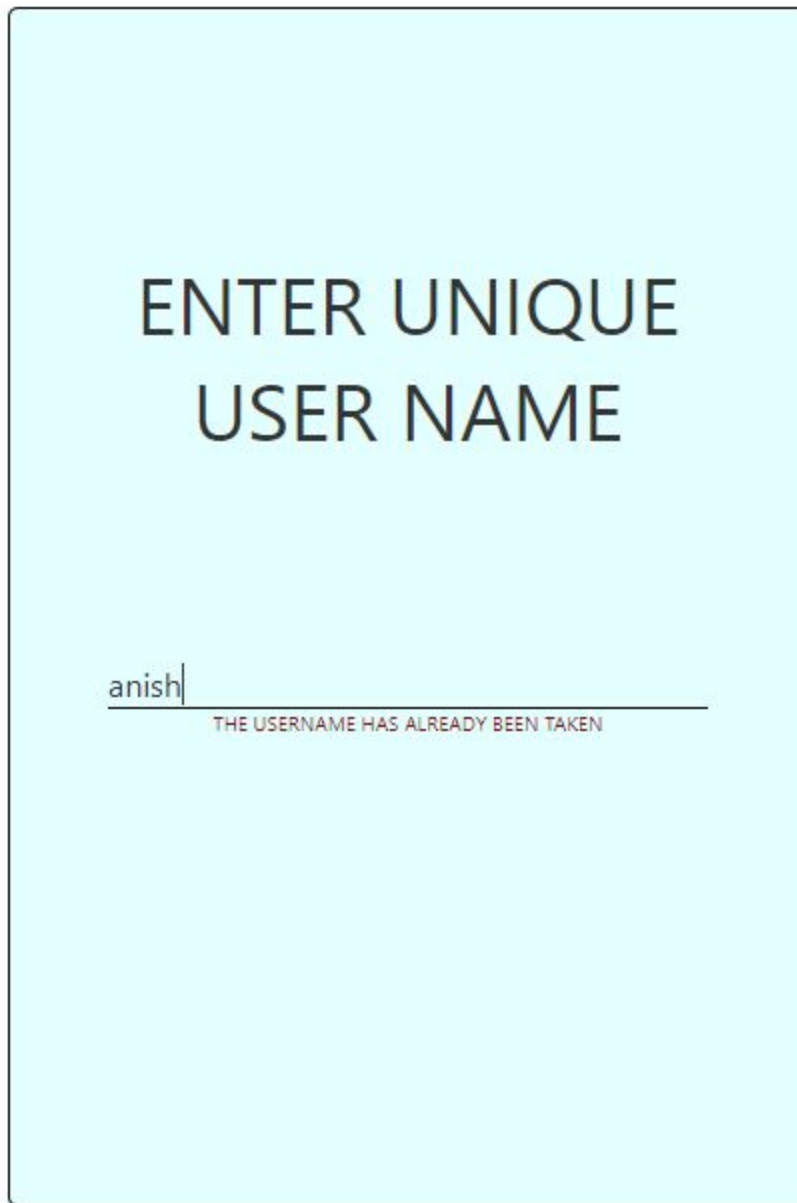
When we open our application we are first greeted with the Login Screen.

A light blue rectangular box representing a login screen. It contains the text "ENTER UNIQUE USER NAME" in a large, dark, sans-serif font, centered. Below the text is a horizontal line with a small vertical tick at the left end, indicating an input field. At the bottom center is a purple rounded rectangle button with the text "JOIN CHAT ROOM" in white, uppercase, sans-serif font.

ENTER UNIQUE
USER NAME

JOIN CHAT ROOM

After that we have to enter our unique username and if the username is already taken, we will be informed and we will not be able to log in with that username.



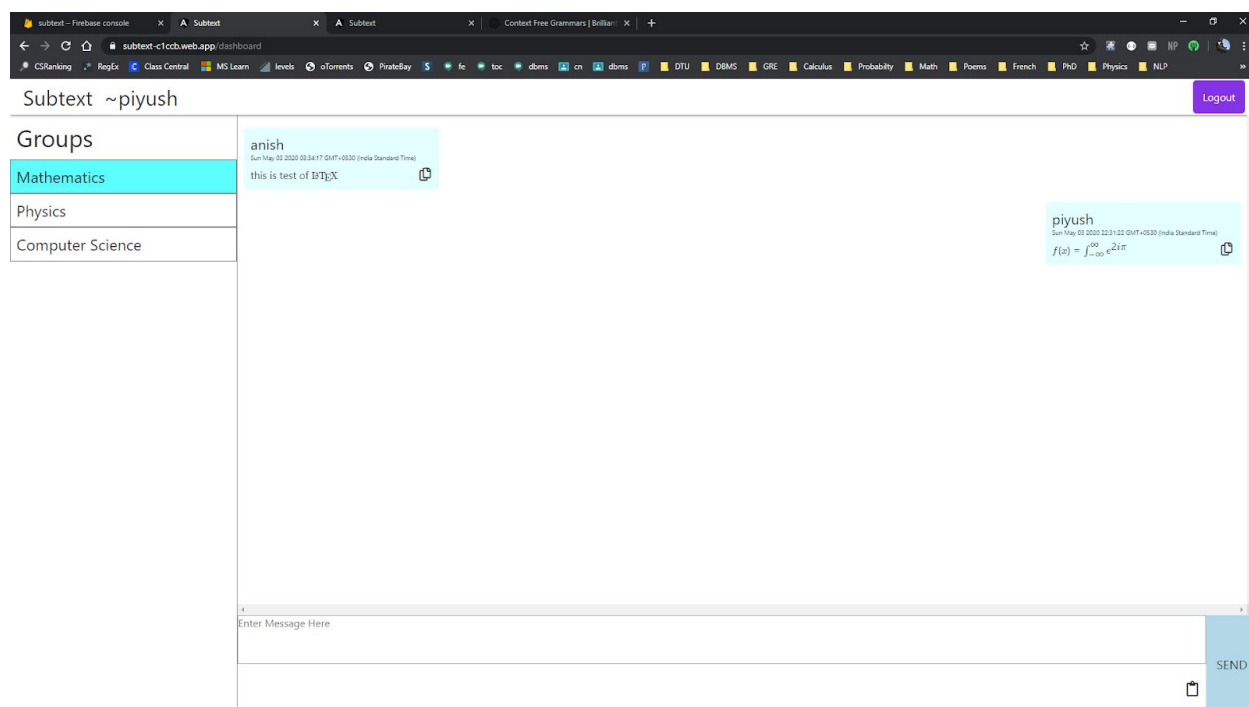
ENTER UNIQUE
USER NAME

anish|

THE USERNAME HAS ALREADY BEEN TAKEN

If the username has been taken, we will have to create/use some other name and then click the join chat room button which will take us into our chatting application.

Once inside we can navigate to any of the groups and read past messages sent by us or anyone else and also send new messages.



LaTeX Support

In the mathematics channel the message box supports LaTeX and we can type any normal message consisting of just text and/or emojis such as:

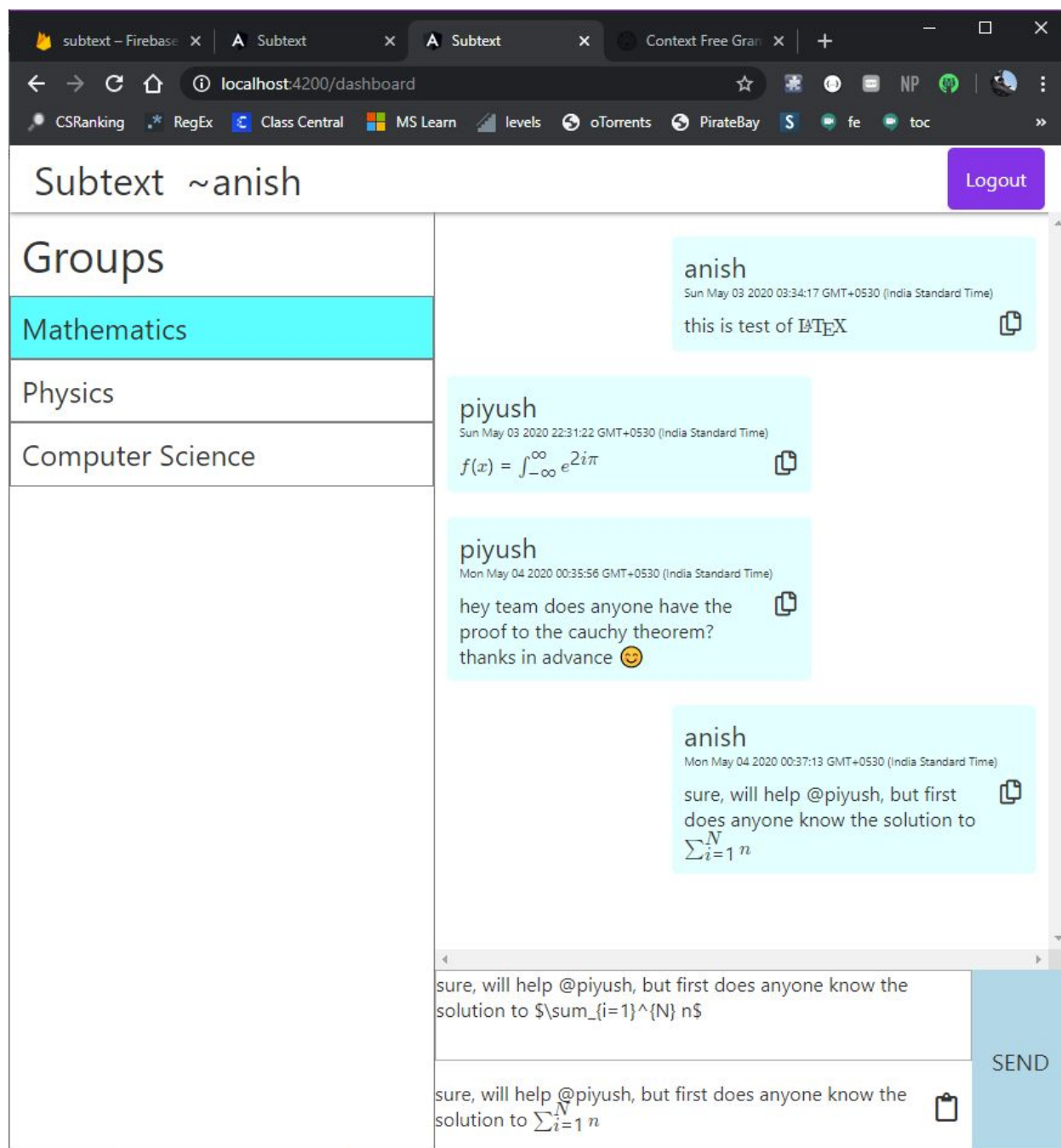
hey team does anyone have the proof to the cauchy theorem? thanks in advance 😊

And we can also compose mathematical messages such as:

sure, will help @piyush, but first does anyone know the solution to $\sum_{i=1}^N n$

sure, will help @piyush, but first does anyone know the solution to $\sum_{i=1}^N n$

Resulting in a group conversation that looks like:



The subtext application also provides a small copy button with every message, so that people don't have to type out all mathematical messages again and again and can simply use the effort that they or others put in while framing previous messages.

After clicking the copy button they can click on clipboard icon in the messaging panel and this will copy not the end result transpiled formatted output, but rather than the inner utf8 text that was used to create the markdown or LaTeX message.

Markdown Support

If the user navigates to the computer science group then the message panel transpiles all valid markdown text into preformatted markdown as follows:

this is <code>_italic_</code> text	this is <code>__bold__</code> text
this is <i>italic</i> text	this is bold text
this is text where by using the parse tree the lexical analyzer will compute that www.google.com is a link	

this is text where by using the parse tree the lexical analyzer will compute that www.google.com is a link

We can even send tables using markdown as:

Tables	Are	Cool
col 1 is	left-aligned	\$1600
col 2 is	amazing	\$12
col 3 is	😊😊	\$1

Tables Are Cool
col 1 is left-aligned \$1600

And this will result in the following output when sent as a message.

Subtext ~piyush

Logout

Groups													
Mathematics													
Physics													
Computer Science	<div><div>piyush</div><div>Mon May 04 2020 00:50:23 GMT+0530 (India Standard Time)</div><div><table><thead><tr><th>Tables</th><th>Are</th><th>Cool</th></tr></thead><tbody><tr><td>col 1 is left-aligned</td><td>\$1600</td><td></td></tr><tr><td>col 2 is amazing</td><td>\$12</td><td></td></tr><tr><td>col 3 is 😊😊</td><td>\$1</td><td></td></tr></tbody></table></div></div>	Tables	Are	Cool	col 1 is left-aligned	\$1600		col 2 is amazing	\$12		col 3 is 😊😊	\$1	
Tables	Are	Cool											
col 1 is left-aligned	\$1600												
col 2 is amazing	\$12												
col 3 is 😊😊	\$1												
	<div><div>Enter Message Here</div><div>SEND</div></div>												

Future Scope

The future tasks are something that accomplishing this semester may be feasible, but the Future scope is something that this application has the potential of becoming or achieving but isn't possible in the short time period of a few months or the remaining time before this semester.

The future scope of this application includes the following features that can be added to this:

For Casual Consumers

Casual consumers (non paying) consumers can simply go to our domain name www.subtext.com and log in using any valid EMail account or through any of the given OAuth interfaces (Facebook OAuth, Google OAuth, GitHub OAuth) After logging in they will have to create a unique username and also pass through basic formalities of creating a password and entering a few security questions. The login process will be very simple and straightforward and will only require the person selecting a unique username (Only part which can potentially take time).

Using a unique username we can overcome a very big hurdle that was brought in by whatsapp. WhatsApp forces that identification happens through the cellular provider number, but it happens very often that there is no cellular provider number or that a signal is unavailable or most common that the number used for creating the account is no longer in use and has been transferred to a third party.

Using a unique username that is not tied down to any other authentication, but just an EMail authentication makes the experience stateless and doesn't pre inforce any knowledge of the user or requirement to use a cellular enabled mobile device for this app.

Having created a username the users can then connect with their family and friends using each other's usernames and use all of the advanced constructs that are available at their disposal.

For Organizations (Future Growth Ideas for product - Optimistic Venture)

Organizations are much more demanding and require much more granular control over users along with authorization rules as well. Organizations also have many products and team internally so the same user in one organization should have the ability to be a part of multiple Groups.

To register an organization a user once logged in can simply click on **Create an Organization** and will be redirected to create a unique organization name that will be used

to manage all the Groups and Channels that will be created within an organization and also the users that will be added in said organization.

Let us say that the organization name be ***organization-name*** and so any person logging into the space of this organization can go to ***organization-name.subtext.com/***. Every organization based on their unique name will receive a sub domain name on the main domain space. Once logged into the organization space, a user can create Groups or chat with individual users.

Group: Group is a collection of channels that can be created by any user.

Channel: Channel is a message communication stream that can be set up inside a group by the group admin or any user based on the authorization rules present in a particular group. A channel provides a message stream for a specialized topic and users present in groups can join various channels based on where they want to contribute and what they wish to be a part of. The addition of a user to a Channel can be further controlled using authorization rules set up by administrators during Group creation and also Channel creation.

Inside a group, a user can communicate in any channel where he is a part of and also communicate with individual users in the same group.

Each Channel inside the group will have a unique name which can not collide with the Global scope of all usernames. Similarly every group name will also be a unique name that can not collide with user names, but can with channel names.

The url for the above groups and channels will be as :

For Chatting inside a particular channel

<https://organization-name.subtext.com/{groupName}/{channelName}>

For chatting with another user in the group

<https://organization-name.subtext.com/user/{userName}>

Markdown Parser Code Snippets

The entire parser consists of many functions that each individually manage the successful conversion of a single markdown construct into an HTML node. A markdown construct here is referring to one single markdown formatting rule that applies e.g bold text or italic text. Here this atomic rule is referring to a single construct and I have created a singular function to handle and manage every such singular construct.

Some functions created to handle these constructs are:

to-heading.ts

This function converts headings given in .md format to html. For example

Heading is a heading in markdown which should be converted into `<h1>Heading</h1>` in html. Similarly for all 6 possible headings in markdown; namely ## Heading 2, ### Heading 3 ... ##### Heading 6. They should all be converted into html using html node tags such as `<h2>Heading 2</h2>` ... `<h6>Heading 6</h6>`.

```
import {toTextBlock} from "./to-text-block";
import {toHashBlock} from "./to-hash-block";

export function toHeading(text: string): string {
  const headerLevelStart = 1;

  const setextRegexH1 = /^(.+)[\t]*\n={2,}[\t]*\n+/gm;
  const setextRegexH2 = /^(.+)[\t]*\n-{2,}[\t]*\n+/gm;

  text = text.replace(setextRegexH1, (wholeMatch, m1) => {

    const spanGamut = toTextBlock(m1);
    const hID = "";
    const hLevel = headerLevelStart;

    const hashBlock = '<h' + hLevel + hID + '>' + spanGamut + '</h' + hLevel + '>';

    return toHashBlock(hashBlock);
  });
}
```

```

text = text.replace(settextRegexH2, (matchFound, m1) => {
  const spanGamut = toTextBlock(m1);
  const hID = "";
  const hLevel = headerLevelStart + 1;
  const hashBlock = '<h' + hLevel + hID + '>' + spanGamut + '</h' + hLevel + '>';
  return toHashBlock(hashBlock);
});

// atx-style headers:
// # Header 1
// ## Header 2
// ## Header 2 with closing hashes ##
// ...
// ##### Header 6
//
const atxStyle = /^(#{1,6})[ \t]+(?:.?) [ \t]*#\n+/gm;

text = text.replace(atxStyle, (wholeMatch, m1, m2) => {
  const span = toTextBlock(m2);
  const hID = "";
  const hLevel = headerLevelStart - 1 + m1.length;
  const header = '<h' + hLevel + hID + '>' + span + '</h' + hLevel + '>';

  return toHashBlock(header);
});

return text;
}

```

to-italics-and-bold.ts

Italics in markdown is represented by `_italics_` and this should be converted into `html` as `italics`. Similarly bold in markdown is represented by `__bold__` and should be converted into `bold`.

```
export function toItalicsAndBold(text: string, options: any): string{
  function parseInside (txt, left, right) {
    return left + txt + right;
  }

  // Parse underscores
  if (options.literalMidWordUnderscores) {
    text = text.replace(/\b__(\S[\s\S]*?)__\b/g, function (wm, txt) {
      return parseInside (txt, '<strong><em>', '</em></strong>');
    });
    text = text.replace(/\b_(\S[\s\S]*?)_\b/g, function (wm, txt) {
      return parseInside (txt, '<strong>', '</strong>');
    });
    text = text.replace(/\b_(\S[\s\S]*?)_\b/g, function (wm, txt) {
      return parseInside (txt, '<em>', '</em>');
    });
  } else {
    text = text.replace(/__(\S[\s\S]*?)__/g, function (wm, m) {
      return (\S$/.test(m)) ? parseInside (m, '<strong><em>', '</em></strong>') : wm;
    });
    text = text.replace(/_(\S[\s\S]*?)_/g, function (wm, m) {
      return (\S$/.test(m)) ? parseInside (m, '<strong>', '</strong>') : wm;
    });
    text = text.replace(/_([\^s_][\s\S]*?)_/g, function (wm, m) {
      // !/^_/.test(m) - test if it doesn't start with _ (since it seems redundant, we removed it)
      return (\S$/.test(m)) ? parseInside (m, '<em>', '</em>') : wm;
    });
  }
}
```

```

// Underscores
text = text.replace(/\*\*(\S[\s\S]*?)\*\*/g, function (wm, m) {
  return (/S$/).test(m) ? parseInside (m, '<strong><em>', '</em></strong>') : wm;
});

text = text.replace(/\*(\S[\s\S]*?)\*/g, function (wm, m) {
  return (/S$/).test(m) ? parseInside (m, '<strong>', '</strong>') : wm;
});

text = text.replace(/\*([\s\S]*?)\*/g, function (wm, m) {
  // !/\*\^[^*]/.test(m) - test if it doesn't start with ** (since it seems redundant, we removed it)
  return (/S$/).test(m) ? parseInside (m, '<em>', '</em>') : wm;
});

return text;
}

```

to-strikethrough.ts

Strikethrough in markdown can be placed as ~~this text has been striked through~~ and should be converted into valid html as this text has been striked through.

```

export function toStrikethrough(text: string): string {
  text = text.replace(/(?:~){2}([\s\S]+?)(?:~){2}/g, (wm, txt) => '<del>' + txt + '</del>');
  return text;
}

```

to-paragraph.ts

```

import {toTextBlock} from "../to-text-block";
import {getHtmlBlock} from "../service/get-html-block";

export function toParagraph(text: string): string {
  let i;
  // Strip leading and trailing lines:
  text = text.replace(/^\n+/g, "");

```

```
text = text.replace(/\n+$/g, "");
```

```
let graft = text.split(/\n{2,}/g);
```

```
let graftOut = [];
```

```
let end = graft.length; // Wrap <p> tags
```

```
for (i = 0; i < end; i++) {
```

```
  let str = graft[i];
```

```
  // if this is an HTML marker, copy it
```

```
  if (str.search(/"([KG])(\d+)\1/g) >= 0) {
```

```
    graftOut.push(str);
```

```
    // test for presence of characters to prevent empty lines being parsed
```

```
    // as paragraphs (resulting in undesired extra empty paragraphs)
```

```
  } else if (str.search(/\S/) >= 0) {
```

```
    str = toTextBlock(str);
```

```
    str = str.replace(/^[ \t]*/g, '<p>');
```

```
    str += '</p>';
```

```
    graftOut.push(str);
```

```
  }
```

```
}
```

```
/** Un-hash HTML blocks */
```

```
end = graftOut.length;
```

```
for (i = 0; i < end; i++) {
```

```
  let blockText = "";
```

```
  let graftOutElement = graftOut[i];
```

```
  let flag = false;
```

```
  // if this is a marker for an html block...
```

```
  // use RegExp.test instead of string.search because of QML bug
```

```
  while (/"/([KG])(\d+)\1/.test(graftOutElement)) {
```

```

    const delimiter = RegExp.$1;
    const num = RegExp.$2;

    if (delimiter === 'K') {
        blockText = getHtmlBlock()[num];
    }

    blockText = blockText.replace(/\$/g, '$$$$'); // Escape any dollar signs

    graftOutElement = graftOutElement.replace(/(\n\n)?"([KG])\d+\2(\n\n)?/, blockText);
    // Check if graftOutElement is a pre->code
    if (/^<pre\b[^\>]*>\s*<code\b[^\>]*>/.test(graftOutElement)) {
        flag = true;
    }
}

graftOut[i] = graftOutElement;
}

text = graftOut.join('\n');
// Strip leading and trailing lines:
text = text.replace(/^\n+/g, "");
text = text.replace(/\n+$/g, "");

return text;
}

```

to-table.ts

Tables in markdown are represented in the following format.

```

| column1      | col2          |
|-----|-----|
| data         | fancy         |
| so cool!!!   | amazing :)    |

```

This should be converted into html as follows:

</table>

```
import {toCodeSpan} from "../to-code-span";
```

```
const tableRegex = /^ {0,3}\| ?\.\n {0,3}\| ?[ \t]*?:[ \t]*?(?:=){2,}[ \t]*?:[ \t]*\| [ \t]*?:[ \t]*?(?:=){2,}[\s\S]+?(?:\n\n | ")/gm;
```

```
function parseStyles (sLine) {
```

```
if (/^[ \t]*--*$/).test(sLine) {
```

```
return ' style="text-align:left;";
```

```
} else if (/^--*[\t]*:[\t]*$/ .test(sLine)) {
```

```

    return ' style="text-align:right;";
  } else if (/^:[ \t]*--*[ \t]*:$/ .test(sLine)) {
    return ' style="text-align:center;";
  } else {
    return "";
  }
}

```

```

const parseHeaders = (header, style, options?: any) => {
  let id = "";
  header = header.trim();

  // support both tablesHeaderId and tableHeaderId due to error in documentation so we don't break
  // backwards compatibility
  if (options.tablesHeaderId || options.tableHeaderId) {
    id = ' id="' + header.replace(/ /g, '_').toLowerCase() + '"';
  }

  header = toTextBlock(header);

  return '<th' + id + style + '>' + header + '</th>\n';
};

```

```

const parseCells = (cell, style) => {
  const subText = toTextBlock(cell);
  return '<td' + style + '>' + subText + '</td>\n';
};

```

```

function buildTable (headers, cells) {
  let i;
  let tb = '<table>\n<thead>\n<tr>\n';

```

```
tblLgn = headers.length;
```

```
for (i = 0; i < tblLgn; ++i) {
```

```
    tb += headers[i];
```

```
}
```

```
tb += '</tr>\n</thead>\n<tbody>\n';
```

```
for (i = 0; i < cells.length; ++i) {
```

```
    tb += '<tr>\n';
```

```
    for (let ii = 0; ii < tblLgn; ++ii) {
```

```
        tb += cells[i][ii];
```

```
    }
```

```
    tb += '</tr>\n';
```

```
}
```

```
tb += '</tbody>\n</table>\n';
```

```
return tb;
```

```
}
```

```
function parseTable (rawTable) {
```

```
    let i, tableLines = rawTable.split('\n');
```

```
    for (i = 0; i < tableLines.length; ++i) {
```

```
        // strip wrong first and last column if wrapped tables are used
```

```
        if (/^ {0,3}\|/.test(tableLines[i])) {
```

```
            tableLines[i] = tableLines[i].replace(/^ {0,3}\|/, "");
```

```
        }
```

```
        if (/^\| [\t]*$/ .test(tableLines[i])) {
```

```
            tableLines[i] = tableLines[i].replace(/^\| [\t]*$/, "");
```

```

    }

    // parse code spans first, but we only support one line code spans

    tableLines[i] = toCodeSpan(tableLines[i]);
  }

  const rawHeaders = tableLines[0].split(' ').map(s => s.trim());
  const rawStyles = tableLines[1].split(' ').map(s => s.trim());
  const rawCells = [];
  const headers = [];
  const styles = [];
  const cells = [];

  tableLines.shift();
  tableLines.shift();

  for (i = 0; i < tableLines.length; ++i) {
    if (tableLines[i].trim() === "") {
      continue;
    }

    rawCells.push(
      tableLines[i]
        .split(' ')
        .map(s => s.trim())
    );
  }

  if (rawHeaders.length < rawStyles.length) {

```



```
    return rawTable;
}

    for (i = 0; i < rawStyles.length; ++i) {
        styles.push(parseStyles(rawStyles[i]));
    }

    for (i = 0; i < rawHeaders.length; ++i) {
        if (isUndefined(styles[i])) {
            styles[i] = "";
        }
        headers.push(parseHeaders(rawHeaders[i], styles[i]));
    }

    for (i = 0; i < rawCells.length; ++i) {
        const row = [];
        for (let ii = 0; ii < headers.length; ++ii) {
            row.push(parseCells(rawCells[i][ii], styles[ii]));
        }
        cells.push(row);
    }

    return buildTable(headers, cells);
}

// parse multi column tables
text = text.replace(tableRegex, parseTable);

// parse one column tables
```

```
text = text.replace(singleColumnRegex, parseTable);
```

```
return text;
```

```
}
```

to-code-block.ts

```
import {toOutDent} from "../to-out-dent";
```

```
import {encodeCode} from "../service/encode-code";
```

```
import {removeTab} from "../remove-tab";
```

```
import {toHashBlock} from "../to-hash-block";
```

```
export function toCodeBlock(text: string): string {
```

```
    // sentinel workarounds for lack of \A and \Z, safari\khtml bug
```

```
    text += '\0';
```

```
    const pattern = /(?:\n\n|^)((?:(?:[ ]{4}|\\t).*\n+)(\n*[ ]{0,3}[^\\t\n]|(?:``0))/g;
```

```
    text = text.replace(pattern, function (wholeMatch, m1, m2) {
```

```
        let codeBlock = m1;
```

```
        let nextChar = m2;
```

```
        let end = '\n';
```

```
        codeBlock = toOutDent(codeBlock);
```

```
        codeBlock = encodeCode(codeBlock);
```

```
        codeBlock = removeTab(codeBlock);
```

```
        codeBlock = codeBlock.replace(/^\n+/g, ""); // trim leading newlines
```

```
        codeBlock = codeBlock.replace(/\n+$/g, ""); // trim trailing newlines
```

```
        codeBlock = '<pre><code>' + codeBlock + end + '</code></pre>';
```

```
        return toHashBlock(codeBlock) + nextChar;
```

```
    });
```

```
// strip sentinel
text = text.replace(/`0/, "");

return text;
}
```

to-block-quotes.ts

```
import {toGitHubCodeBlock} from "./to-github-code-block";
import {toTextBlock} from "./to-text-block";
import {toHashBlock} from "./to-hash-block";

export function toBlockQuotes(text: string): string {

  // add a couple extra lines after the text
  text = text + '\n\n';

  let regex = /^(^ {0,3}>[ \t]?.\n(.+\n)*\n*)+/gm;

  text = text.replace(regex, (blockQuote) => {
    blockQuote = blockQuote.replace(/^[ \t]*>[ \t]?/gm, ""); // trim one level of quoting

    blockQuote = blockQuote.replace(/`0/g, "");

    // trim whitespace-only lines
    blockQuote = blockQuote.replace(/^[ \t]+$ /gm, "");
    blockQuote = toGitHubCodeBlock(blockQuote);
    blockQuote = toTextBlock(blockQuote);

    blockQuote = blockQuote.replace(/(^|\n)/g, '$1 ');

    // These leading spaces screw with <pre> content, so we need to fix that:
    blockQuote = blockQuote.replace(/(\s*<pre>[^\r]+?<\pre>)/gm, function (wholeMatch, m1) {
```

```

    let pre = m1;

    // attacklab: hack around Konqueror 3.5.4 bug:
    pre = pre.replace(/^ /mg, '0');
    pre = pre.replace(/`0/g, "");
    return pre;
  });

  return toHashBlock('<blockquote>\n' + blockQuote + '\n</blockquote>');
});

return text;
}

```

to-emoji.ts

```

import {emojis} from "./emojis";

export function toEmoji(text: string): string {
  const emojiRgx = /:([\S]+?):/g;

  text = text.replace(emojiRgx, (wm, emojiCode) => {
    if (emojis.hasOwnProperty(emojiCode)) {
      return emojis[emojiCode];
    }
    return wm;
  });

  return text;
}

```

emojis.ts

```

export const emojis = {
  '+1': '\ud83d\udc4d',

```

'-1': '\ud83d\udc4e',
'100': '\ud83d\udcaf',
'1234': '\ud83d\udd22',
'1st_place_medal': '\ud83e\udd47',
'2nd_place_medal': '\ud83e\udd48',
'3rd_place_medal': '\ud83e\udd49',
'8ball': '\ud83c\udfb1',
'a': '\ud83c\udd70\ufe0f',
'ab': '\ud83c\udd8e',
'abc': '\ud83d\udd24',
'abcd': '\ud83d\udd21',
'accept': '\ud83c\ude51',
'aerial_tramway': '\ud83d\udea1',
'airplane': '\u2708\ufe0f',
'alarm_clock': '\u23f0',
'alembic': '\u2697\ufe0f',
'alien': '\ud83d\udc7d',
'ambulance': '\ud83d\ude91',
'amphora': '\ud83c\udffa',
'anchor': '\u2693\ufe0f',
'angel': '\ud83d\udc7c',
'anger': '\ud83d\udca2',
'angry': '\ud83d\ude20',
'anguished': '\ud83d\ude27',
'ant': '\ud83d\udc1c',
'apple': '\ud83c\udf4e',
'aquarius': '\u2652\ufe0f',
'aries': '\u2648\ufe0f',
'arrow_backward': '\u25c0\ufe0f',
'arrow_double_down': '\u23ec',
'arrow_double_up': '\u23eb',
'arrow_down': '\u2b07\ufe0f',

```

'arrow_down_small': '\ud83d\udd3d',
'arrow_forward': '\u25b6\ufe0f',
'arrow_heading_down': '\u2935\ufe0f',
'arrow_heading_up': '\u2934\ufe0f',
'arrow_left': '\u2b05\ufe0f',
'arrow_lower_left': '\u2199\ufe0f',
'arrow_lower_right': '\u2198\ufe0f',
'arrow_right': '\u27a1\ufe0f',
'arrow_right_hook': '\u21aa\ufe0f',
'arrow_up': '\u2b06\ufe0f',
'arrow_up_down': '\u2195\ufe0f',
'arrow_up_small': '\ud83d\udd3c',
'arrow_upper_left': '\u2196\ufe0f',
'arrow_upper_right': '\u2197\ufe0f',
'arrows_clockwise': '\ud83d\udd03',
'arrows_counterclockwise': '\ud83d\udd04',
'art': '\ud83c\udfa8',
'articulated_lorry': '\ud83d\ude9b',
'artificial_satellite': '\ud83d\uddef0',
'astonished': '\ud83d\ude32',
'athletic_shoe': '\ud83d\udc5f',
'atm': '\ud83c\udfe7',
'atom_symbol': '\u269b\ufe0f',
'avocado': '\ud83e\udd51',
'b': '\ud83c\udd71\ufe0f',
'baby': '\ud83d\udc76',
'baby_bottle': '\ud83c\udf7c',
'baby_chick': '\ud83d\udc24'
};

```

to-list.ts

```
import {toOutDent} from "../to-out-dent";
```

```

import {toGitHubCodeBlock} from "./to-github-code-block";
import {toTextBlock} from "./to-text-block";
import {hashHtmlBlock} from "../service/hash-html-block";
import {toParagraph} from "./to-paragraph";

export function toList(text: string): string{

  /**
   * Process the contents of a single ordered or unordered list, splitting it
   * into individual list items.
   */

  const processListItems = (listStr: string, trimTrailing): string => {

    // trim trailing blank lines:
    listStr = listStr.replace(/\n{2,}$/, '\n');

    listStr += '\n0';

    let rgx = /(\n)?(^ {0,3})([*+-]|\d+\.)([ \t]+((\[(x|X| )?)?\[ \t]*[^\r]+?(\\n{1,2}))?(=\n*(\n0|
{0,3})([*+-]|\d+\.)([ \t]+))/gm;

    let isParagraphed = (/\\n[ \t]*\\n(?:\n0)/.test(listStr));

    listStr = listStr.replace(rgx, (wholeMatch, m1, m2, m3, m4, taskbtn, checked) => {
      checked = (checked && checked.trim() !== "");

      let item = toOutDent(m4);
      let bulletStyle = "";

      // Support for github tasklists
      if (taskbtn) {
        bulletStyle = ' class="task-list-item" style="list-style-type: none;';
        item = item.replace(/^([ \t]*\\[(x|X| )?)?/m, function () {

```

```

    let otp = '<input type="checkbox" disabled style="margin: 0px 0.35em 0.25em -1.6em;
vertical-align: middle;";
    if (checked) {
        otp += ' checked';
    }
    otp += '>';
    return otp;
});
}

```

```

item = item.replace(/^[^*+]|\\d\\.)(\\t|+\\S\\n )*/g, wm2 => ' ' + wm2);

```

*// SPECIAL CASE: an heading followed by a paragraph of text that is not separated by a double
newline*

// or/nor indented. ex:

//

// - # foo

// bar is great

//

// While this does now follow the spec per se, not allowing for this might cause confusion since

// header blocks don't need double newlines after

```

if (/^#+.+\\n.+/.test(item)) {
    item = item.replace(/^(#+.+)$/m, '$1\\n');
}

```

// m1 - Leading line or

// Has a double return (multi paragraph)

```

if (m1 || (item.search(/\\n{2,}/) > -1)) {
    item = toGitHubCodeBlock(item);
    item = toTextBlock(item);
} else {

```



```

    // Recursion for sub-lists:
    item = toList(item);
    item = item.replace(/\n$/, ""); //.chomp(item)
    item = hashHtmlBlock(item);

    // Collapse double linebreaks
    item = item.replace(/\n\n+/g, '\n\n');

    if (isParagraphed) {
        item = toParagraph(item);
    } else {
        item = toTextBlock(item);
    }
}

// now we need to remove the marker ("A")
item = item.replace("A", "");
// we can finally wrap the line in list item tags
item = '<li' + bulletStyle + '>' + item + '</li>\n';

return item;
});

// strip sentinel
listStr = listStr.replace(/"0/g, "");

if (trimTrailing) {
    listStr = listStr.replace(/\s+$/, "");
}

return listStr;
};

```

```

function styleStartNumber (list, listType) {
  // check if ol and starts by a number different than 1
  if (listType === 'ol') {
    const res = list.match(/^*(\d+)\./);
    if (res && res[1] !== '1') {
      return ' start="' + res[1] + '"';
    }
  }
  return "";
}

/**
 * Check and parse consecutive lists
 */

function parseConsecutiveLists (list: string, listType: string, trimTrailing: boolean): string {
  // check if we caught 2 or more consecutive lists by mistake
  // we use the counterRgx, meaning if listType is UL we look for OL and vice versa
  let olRgx = /^?\d+\.[ \t]/gm;
  let ulRgx = /^?[*+~][ \t]/gm;
  let counterRgx = (listType === 'ul') ? olRgx : ulRgx;
  let result = "";

  if (list.search(counterRgx) === -1) {
    const style = styleStartNumber(list, listType);
    result = '\n\n<' + listType + style + '>\n' + processListItems(list, !!trimTrailing) + '</' + listType + '>\n';
  } else (function parseCL(txt) {
    const pos = txt.search(counterRgx);
    const style = styleStartNumber(list, listType);
    if (pos !== -1) {

```

```

    result += '\n\n<' + listType + style + '>\n' + processListItems(txt.slice(0, pos), !!trimTrailing) +
    '</' + listType + '>\n';

    // invert counterType and listType
    listType = (listType === 'ul' ? 'ol' : 'ul');
    counterRgx = (listType === 'ul' ? olRgx : ulRgx);

    //recurse
    parseCL(txt.slice(pos));
  } else {
    result += '\n\n<' + listType + style + '>\n' + processListItems(txt, !!trimTrailing) + '</' + listType
    + '>\n';
  }
})(list);

return result;
}

// Start of list parsing
const childListRegex = /^( {0,3}([*+-] | \d+[\.]) [ \t]+)[^\r]+?(`0 | \n{2,}(?=\S)(?![ \t]*([*+-] | \d+[\.]) [ \t]+)))/gm;
const mainListRegex = /(\n\n | ^\n?)(( {0,3}([*+-] | \d+[\.]) [ \t]+)[^\r]+?(`0 | \n{2,}(?=\S)(?![ \t]*([*+-] | \d+[\.]) [ \t]+)))/gm;

text += '`0';

// strip sentinel
text = text.replace(/`0/, "");
return text;
}

```

LaTeX Parser Code Snippets

Just as we implemented singular granular functions for the markdown parser that convert an atomic construct of markdown to LaTeX, similarly we have created a **latex-to-html-parser** that is a function library replete with multiple functions that each parse and transpile an atomic LaTeX construct into valid HTML.

Unlike the markdown parser for which there exists HTML elements corresponding to each and every markdown feature, that is not the case with the LaTeX language and a special set of fonts has been compiled and collated together to bring to `html` in `svg` (scalable vector format) all the missing symbols and language features.

Svg format guarantees that the drawn vector will look good on any and all screen sizes and resolutions. It will look just as if LaTeX were being rendered in a proper TeX based editor into pdf.

Some atomic conversion functions are as follows:

`sqrt.js`

```
import defineFunction from "../defineFunction";
import buildCommon from "../buildCommon";
import mathMLTree from "../mathMLTree";
import delimiter from "../delimiter";
import Style from "../Style";
```

```
import * as html from "../buildHTML";
import * as mml from "../buildMathML";
```

```
defineFunction({
  type: "sqrt",
  names: ["\\sqrt"],
  props: {
    numArgs: 1,
    numOptionalArgs: 1,
  },
  handler({parser}, args, optArgs) {
```

```

const index = optArgs[0];
const body = args[0];
return {
  type: "sqrt",
  mode: parser.mode,
  body,
  index,
};
},
htmlBuilder(group, options) {
  // Square roots are handled in the TeXbook pg. 443, Rule 11.

  // First, we do the same steps as in overline to build the inner group
  // and line
  let inner = html.buildGroup(group.body, options.havingCrampedStyle());
  if (inner.height === 0) {
    // Render a small surd.
    inner.height = options.fontMetrics().xHeight;
  }

  // Some groups can return document fragments. Handle those by wrapping
  // them in a span.
  inner = buildCommon.wrapFragment(inner, options);

  // Calculate the minimum size for the \surd delimiter
  const metrics = options.fontMetrics();
  const theta = metrics.defaultRuleThickness;

  let phi = theta;
  if (options.style.id < Style.TEXT.id) {
    phi = options.fontMetrics().xHeight;
  }
}

```

```
// Calculate the clearance between the body and line
```

```
let lineClearance = theta + phi / 4;
```

```
const minDelimiterHeight = (inner.height + inner.depth +  
  lineClearance + theta);
```

```
// Create a sqrt SVG of the required minimum size
```

```
const {span: img, ruleWidth, advanceWidth} =  
  delimiter.sqrtImage(minDelimiterHeight, options);
```

```
const delimDepth = img.height - ruleWidth;
```

```
// Adjust the clearance based on the delimiter size
```

```
if (delimDepth > inner.height + inner.depth + lineClearance) {  
  lineClearance =  
    (lineClearance + delimDepth - inner.height - inner.depth) / 2;  
}
```

```
// Shift the sqrt image
```

```
const imgShift = img.height - inner.height - lineClearance - ruleWidth;
```

```
inner.style.paddingLeft = advanceWidth + "em";
```

```
// Overlay the image and the argument.
```

```
const body = buildCommon.makeVList({  
  positionType: "firstBaseline",  
  children: [  
    {type: "elem", elem: inner, wrapperClasses: ["svg-align"]},  
    {type: "kern", size: -(inner.height + imgShift)},  
    {type: "elem", elem: img},  
    {type: "kern", size: ruleWidth},
```

```

    ],
    }, options);

    if (!group.index) {
        return buildCommon.makeSpan(["mord", "sqrt"], [body], options);
    } else {
        // Handle the optional root index

        // The index is always in scriptscript style
        const newOptions = options.havingStyle(Style.SCRIPTSCRIPT);
        const rootm = html.buildGroup(group.index, newOptions, options);

        // The amount the index is shifted by. This is taken from the TeX
        // source, in the definition of `r@@t`.
        const toShift = 0.6 * (body.height - body.depth);

        // Build a VList with the superscript shifted up correctly
        const rootVList = buildCommon.makeVList({
            positionType: "shift",
            positionData: -toShift,
            children: [{type: "elem", elem: rootm}],
        }, options);
        // Add a class surrounding it so we can add on the appropriate
        // kerning
        const rootVListWrap = buildCommon.makeSpan(["root"], [rootVList]);

        return buildCommon.makeSpan(["mord", "sqrt"],
            [rootVListWrap, body], options);
    }
},

mathmlBuilder(group, options) {
    const {body, index} = group;

```

```

    return index ?
      new mathMLTree.MathNode(
        "mroot", [
          mml.buildGroup(body, options),
          mml.buildGroup(index, options),
        ]) :
      new mathMLTree.MathNode(
        "msqrt", [mml.buildGroup(body, options)]);
  },
});

```

underline.js

```

import defineFunction from "../defineFunction";
import buildCommon from "../buildCommon";
import mathMLTree from "../mathMLTree";

```

```

import * as html from "../buildHTML";
import * as mml from "../buildMathML";

```

```

defineFunction({
  type: "underline",
  names: ["\\underline"],
  props: {
    numArgs: 1,
    allowedInText: true,
  },
  handler({parser}, args) {
    return {
      type: "underline",
      mode: parser.mode,
      body: args[0],
    };
  }
});

```



```

},
htmlBuilder(group, options) {
  // Underlines are handled in the TeXbook pg 443, Rule 10.
  // Build the inner group.
  const innerGroup = html.buildGroup(group.body, options);

  // Create the line to go below the body
  const line = buildCommon.makeLineSpan("underline-line", options);

  // Generate the vlist, with the appropriate kerns
  const defaultRuleThickness = options.fontMetrics().defaultRuleThickness;
  const vlist = buildCommon.makeVList({
    positionType: "top",
    positionData: innerGroup.height,
    children: [
      {type: "kern", size: defaultRuleThickness},
      {type: "elem", elem: line},
      {type: "kern", size: 3 * defaultRuleThickness},
      {type: "elem", elem: innerGroup},
    ],
  }, options);

  return buildCommon.makeSpan(["mord", "underline"], [vlist], options);
},
mathmlBuilder(group, options) {
  const operator = new mathMLTree.MathNode(
    "mo", [new mathMLTree.TextNode("\u203e")]);
  operator.setAttribute("stretchy", "true");

  const node = new mathMLTree.MathNode(
    "munder",
    [mml.buildGroup(group.body, options), operator]);

```

```

node.setAttribute("accentunder", "true");

return node;
},
});

```

arrow.js

```

import defineFunction from "../defineFunction";
import buildCommon from "../buildCommon";
import mathMLTree from "../mathMLTree";
import stretchy from "../stretchy";

import * as html from "../buildHTML";
import * as mml from "../buildMathML";

import type {ParseNode} from "../parseNode";

// Helper function
const paddedNode = group => {
  const node = new mathMLTree.MathNode("mpadded", group ? [group] : []);
  node.setAttribute("width", "+0.6em");
  node.setAttribute("lspace", "0.3em");
  return node;
};

// Stretchy arrows with an optional argument
defineFunction({
  type: "xArrow",
  names: [
    "\\xleftarrow", "\\xrightarrow", "\\xLeftarrow", "\\xRightarrow",
    "\\xleftrightharrow", "\\xLefttrightharrow", "\\xhookleftarrow",
    "\\xhookrightarrow", "\\xmapsto", "\\xrightarrowpoondown",

```

```

    "\\xrightarrow", "\\xleftarrow", "\\xleftrightarrow",
    "\\xrightarrow", "\\xleftarrow", "\\xlongequal",
    "\\twoheadrightarrow", "\\twoheadleftarrow", "\\xrightarrow",
    // The next 3 functions are here to support the mhchem extension.
    // Direct use of these functions is discouraged and may break someday.
    "\\xrightarrow", "\\xrightleftharpoons", "\\xleftequilibrium",
  ],
  props: {
    numArgs: 1,
    numOptionalArgs: 1,
  },
  handler({parser, funcName, args, optArgs}) {
    return {
      type: "xArrow",
      mode: parser.mode,
      label: funcName,
      body: args[0],
      below: optArgs[0],
    };
  },
  // Flow is unable to correctly infer the type of `group`, even though it's
  // unambiguously determined from the passed-in `type` above.
  htmlBuilder(group: ParseNode<"xArrow">, options) {
    const style = options.style;

    // Build the argument groups in the appropriate style.
    // Ref: amsmath.dtx: \hbox{$\scriptstyle\mkern#3mu{#6}\mkern#4mu$}%

    // Some groups can return document fragments. Handle those by wrapping
    // them in a span.
    let newOptions = options.havingStyle(style.sup());
    const upperGroup = buildCommon.wrapFragment(

```

```

    html.buildGroup(group.body, newOptions, options), options);
    upperGroup.classes.push("x-arrow-pad");

    let lowerGroup;
    if (group.below) {
        // Build the lower group
        newOptions = options.havingStyle(style.sub());
        lowerGroup = buildCommon.wrapFragment(
            html.buildGroup(group.below, newOptions, options), options);
        lowerGroup.classes.push("x-arrow-pad");
    }

    const arrowBody = stretchy.svgSpan(group, options);

    // Re shift: Note that stretchy.svgSpan returned arrowBody.depth = 0.
    // The point we want on the math axis is at 0.5 * arrowBody.height.
    const arrowShift = -options.fontMetrics().axisHeight +
        0.5 * arrowBody.height;

    // 2 mu kern. Ref: amsmath.dtx: #7\if0#2\else\mkern#2mu\fi
    let upperShift = -options.fontMetrics().axisHeight
        - 0.5 * arrowBody.height - 0.111; // 0.111 em = 2 mu
    if (upperGroup.depth > 0.25 || group.label === "\\xleftetequilibrium") {
        upperShift -= upperGroup.depth; // shift up if depth encroaches
    }

    // Generate the vlist
    let vlist;
    if (lowerGroup) {
        const lowerShift = -options.fontMetrics().axisHeight
            + lowerGroup.height + 0.5 * arrowBody.height
            + 0.111;
        vlist = buildCommon.makeVList({

```

```

    positionType: "individualShift",
    children: [
      {type: "elem", elem: upperGroup, shift: upperShift},
      {type: "elem", elem: arrowBody, shift: arrowShift},
      {type: "elem", elem: lowerGroup, shift: lowerShift},
    ],
  }, options);
} else {
  vlist = buildCommon.makeVList({
    positionType: "individualShift",
    children: [
      {type: "elem", elem: upperGroup, shift: upperShift},
      {type: "elem", elem: arrowBody, shift: arrowShift},
    ],
  }, options);
}

// $FlowFixMe: Replace this with passing "svg-align" into makeVList.
vlist.children[0].children[0].children[1].classes.push("svg-align");

return buildCommon.makeSpan(["mrel", "x-arrow"], [vlist], options);
},
mathmlBuilder(group, options) {
  const arrowNode = stretchy.mathMLnode(group.label);
  let node;

  if (group.body) {
    const upperNode = paddedNode(mml.buildGroup(group.body, options));
    if (group.below) {
      const lowerNode = paddedNode(mml.buildGroup(group.below, options));
      node = new mathMLTree.MathNode(
        "munderover", [arrowNode, lowerNode, upperNode]

```

```

    );
  } else {
    node = new mathMLTree.MathNode("mover", [arrowNode, upperNode]);
  }
} else if (group.below) {
  const lowerNode = paddedNode(mml.buildGroup(group.below, options));
  node = new mathMLTree.MathNode("munder", [arrowNode, lowerNode]);
} else {
  // This should never happen.
  // Parser.js throws an error if there is no argument.
  node = paddedNode();
  node = new mathMLTree.MathNode("mover", [arrowNode, node]);
}
return node;
},
});

```

math.js

This function is for recognizing when the delimiter $\$$ has been used and switching over to math mode and parsing the text inside the $\$ \dots \$$ using the parser and the functions given.

```

import defineFunction from "../defineFunction";
import ParseError from "../ParseError";

```

// Switching from text mode back to math mode

```

defineFunction({
  type: "styling",
  names: ["\\(", "$"],
  props: {
    numArgs: 0,
    allowedInText: true,
    allowedInMath: false,
  },

```

```
handler({funcName, parser}, args) {
  const outerMode = parser.mode;
  parser.switchMode("math");
  const close = (funcName === "\\(" ? "\\)" : "$");
  const body = parser.parseExpression(false, close);
  parser.expect(close);
  parser.switchMode(outerMode);
  return {
    type: "styling",
    mode: parser.mode,
    style: "text",
    body,
  };
},
});

// Check for extra closing math delimiters
defineFunction({
  type: "text", // Doesn't matter what this is.
  names: ["\\(", "\\)"],
  props: {
    numArgs: 0,
    allowedInText: true,
    allowedInMath: false,
  },
  handler(context, args) {
    throw new ParseError(`Mismatched ${context.funcName}`);
  },
});
```

buildHTML.js

This is the main function in the project which takes in the parse tree and then outputs the built HTML sanitized and minified.

```
// @flow

/**
 * This file does the main work of building a domTree structure from a parse
 * tree. The entry point is the `buildHTML` function, which takes a parse tree.
 * Then, the buildExpression, buildGroup, and various groupBuilders functions
 * are called, to produce a final HTML tree.
 */

import ParseError from "./ParseError";
import Style from "./Style";
import buildCommon from "./buildCommon";
import {Span, Anchor} from "./domTree";
import utils from "./utils";
import {spacings, tightSpacings} from "./spacingData";
import {_htmlGroupBuilders as groupBuilders} from "./defineFunction";
import {DocumentFragment} from "./tree";

import type Options from "./Options";
import type {AnyParseNode} from "./parseNode";
import type {HtmlDomNode, DomSpan} from "./domTree";

const makeSpan = buildCommon.makeSpan;

// Binary atoms (first class `mbin`) change into ordinary atoms (`mord`)
// depending on their surroundings. See TeXbook pg. 442-446, Rules 5 and 6,
// and the text before Rule 19.

const binLeftCanceller = ["leftmost", "mbin", "mopen", "mrel", "mop", "mpunct"];
const binRightCanceller = ["rightmost", "mrel", "mclose", "mpunct"];
```



```
const styleMap = {
  "display": Style.DISPLAY,
  "text": Style.TEXT,
  "script": Style.SCRIPT,
  "scriptscript": Style.SCRIPTSCRIPT,
};
```

```
type Side = "left" | "right";
```

```
const DomEnum = {
  mord: "mord",
  mop: "mop",
  mbin: "mbin",
  mrel: "mrel",
  mopen: "mopen",
  mclose: "mclose",
  mpunct: "mpunct",
  minner: "minner",
};
```

```
type DomType = $Keys<typeof DomEnum>;
```

```
/**
```

```
 * Take a list of nodes, build them in order, and return a list of the built
 * nodes. documentFragments are flattened into their contents, so the
 * returned list contains no fragments. `isRealGroup` is true if `expression`
 * is a real group (no atoms will be added on either side), as opposed to
 * a partial group (e.g. one created by \color). `surrounding` is an array
 * consisting type of nodes that will be added to the left and right.
```

```
*/
```

```
export const buildExpression = function(
  expression: AnyParseNode[],
```

```

options: Options,
isRealGroup: boolean,
surrounding: [?DomType, ?DomType] = [null, null],
): HtmlDomNode[] {
  // Parse expressions into `groups`.
  const groups: HtmlDomNode[] = [];
  for (let i = 0; i < expression.length; i++) {
    const output = buildGroup(expression[i], options);
    if (output instanceof DocumentFragment) {
      const children: $ReadOnlyArray<HtmlDomNode> = output.children;
      groups.push(...children);
    } else {
      groups.push(output);
    }
  }

  // If `expression` is a partial group, let the parent handle spacings
  // to avoid processing groups multiple times.
  if (!isRealGroup) {
    return groups;
  }

  let glueOptions = options;
  if (expression.length === 1) {
    const node = expression[0];
    if (node.type === "sizing") {
      glueOptions = options.havingSize(node.size);
    } else if (node.type === "styling") {
      glueOptions = options.havingStyle(styleMap[node.style]);
    }
  }
}

```

```
// Dummy spans for determining spacings between surrounding atoms.
```

```
// If `expression` has no atoms on the left or right, class "leftmost"
```

```
// or "rightmost", respectively, is used to indicate it.
```

```
const dummyPrev = makeSpan([surrounding[0] || "leftmost"], [], options);
```

```
const dummyNext = makeSpan([surrounding[1] || "rightmost"], [], options);
```

```
// TODO: These code assumes that a node's math class is the first element
```

```
// of its `classes` array. A later cleanup should ensure this, for
```

```
// instance by changing the signature of `makeSpan`.
```

```
// Before determining what spaces to insert, perform bin cancellation.
```

```
// Binary operators change to ordinary symbols in some contexts.
```

```
traverseNonSpaceNodes(groups, (node, prev) => {
```

```
  const prevType = prev.classes[0];
```

```
  const type = node.classes[0];
```

```
  if (prevType === "mbin" && utils.contains(binRightCanceller, type)) {
```

```
    prev.classes[0] = "mord";
```

```
  } else if (type === "mbin" && utils.contains(binLeftCanceller, prevType)) {
```

```
    node.classes[0] = "mord";
```

```
  }
```

```
}, {node: dummyPrev}, dummyNext);
```

```
traverseNonSpaceNodes(groups, (node, prev) => {
```

```
  const prevType = getTypeOfDomTree(prev);
```

```
  const type = getTypeOfDomTree(node);
```

```
// 'mtight' indicates that the node is script or scriptscript style.
```

```
const space = prevType && type ? (node.hasClass("mtight")
```

```
  ? tightSpacings[prevType][type]
```

```
  : spacings[prevType][type]) : null;
```

```
if (space) { // Insert glue (spacing) after the `prev`.
```

```
  return buildCommon.makeGlue(space, glueOptions);
```

```

    }
    }, {node: dummyPrev}, dummyNext);

    return groups;
};

```

*// Depth-first traverse non-space `nodes`, calling `callback` with the current and
 // previous node as arguments, optionally returning a node to insert after the
 // previous node. `prev` is an object with the previous node and `insertAfter`
 // function to insert after it. `next` is a node that will be added to the right.
 // Used for bin cancellation and inserting spacings.*

```

const traverseNonSpaceNodes = function(
  nodes: HtmlDomNode[],
  callback: (HtmlDomNode, HtmlDomNode) => ?HtmlDomNode,
  prev: { |
    node: HtmlDomNode,
    insertAfter?: HtmlDomNode => void,
  } |,
  next: ?HtmlDomNode,
) {
  if (next) { // temporarily append the right node, if exists
    nodes.push(next);
  }

  let i = 0;
  for (; i < nodes.length; i++) {
    const node = nodes[i];
    const partialGroup = checkPartialGroup(node);
    if (partialGroup) { // Recursive DFS
      // $FlowFixMe: make nodes a $ReadOnlyArray by returning a new array
      traverseNonSpaceNodes(partialGroup.children, callback, prev);
      continue;
    }
  }
}

```

```

// Ignore explicit spaces (e.g., \;, \,) when determining what implicit
// spacing should go between atoms of different classes
if (node.classes[0] === "mspace") {
  continue;
}

const result = callback(node, prev.node);
if (result) {
  if (prev.insertAfter) {
    prev.insertAfter(result);
  } else { // insert at front
    nodes.unshift(result);
    i++;
  }
}

prev.node = node;
prev.insertAfter = (index => n => {
  nodes.splice(index + 1, 0, n);
  i++;
})(i);
}
if (next) {
  nodes.pop();
}
};

// Check if given node is a partial group, i.e., does not affect spacing around.
const checkPartialGroup = function(
  node: HtmlDomNode,
): ?(DocumentFragment<HtmlDomNode> | Anchor | DomSpan) {

```

```

    if (node instanceof DocumentFragment || node instanceof Anchor
        || (node instanceof Span && node.hasClass("enclosing"))) {
        return node;
    }
    return null;
};

```

// Return the outermost node of a domTree.

```

const getOutermostNode = function(
    node: HtmlDomNode,
    side: Side,
): HtmlDomNode {
    const partialGroup = checkPartialGroup(node);
    if (partialGroup) {
        const children = partialGroup.children;
        if (children.length) {
            if (side === "right") {
                return getOutermostNode(children[children.length - 1], "right");
            } else if (side === "left") {
                return getOutermostNode(children[0], "left");
            }
        }
    }
    return node;
};

```

// Return math atom class (mclass) of a domTree.

// If `side` is given, it will get the type of the outermost node at given side.

```

export const getTypeOfDomTree = function(
    node: ?HtmlDomNode,
    side: ?Side,
): ?DomType {

```

```

    if (!node) {
        return null;
    }
    if (side) {
        node = getOutermostNode(node, side);
    }
    // This makes a lot of assumptions as to where the type of atom
    // appears. We should do a better job of enforcing this.
    return DomEnum[node.classes[0]] || null;
};

export const makeNullDelimiter = function(
    options: Options,
    classes: string[],
): DomSpan {
    const moreClasses = ["nullDelimiter"].concat(options.baseSizingClasses());
    return makeSpan(classes.concat(moreClasses));
};

/**
 * buildGroup is the function that takes a group and calls the correct groupType
 * function for it. It also handles the interaction of size and style changes
 * between parents and children.
 */
export const buildGroup = function(
    group: ?AnyParseNode,
    options: Options,
    baseOptions?: Options,
): HtmlDomNode {
    if (!group) {
        return makeSpan();
    }

```

```

    if (groupBuilders[group.type]) {
        // Call the groupBuilders function
        // $FlowFixMe
        let groupNode: HtmlDomNode = groupBuilders[group.type](group, options);

        // If the size changed between the parent and the current group, account
        // for that size difference.
        if (baseOptions && options.size !== baseOptions.size) {
            groupNode = makeSpan(options.sizingClasses(baseOptions),
                [groupNode], options);

            const multiplier =
                options.sizeMultiplier / baseOptions.sizeMultiplier;

            groupNode.height *= multiplier;
            groupNode.depth *= multiplier;
        }

        return groupNode;
    } else {
        throw new ParseError(
            "Got group of unknown type: '" + group.type + "'");
    }
};

/**
 * Combine an array of HTML DOM nodes (e.g., the output of `buildExpression`)
 * into an unbreakable HTML node of class .base, with proper struts to
 * guarantee correct vertical extent. `buildHTML` calls this repeatedly to
 * make up the entire expression as a sequence of unbreakable units.
 */

```



```

function buildHTMLUnbreakable(children, options) {
  // Compute height and depth of this chunk.
  const body = makeSpan(["base"], children, options);

  // Add strut, which ensures that the top of the HTML element falls at
  // the height of the expression, and the bottom of the HTML element
  // falls at the depth of the expression.
  const strut = makeSpan(["strut"]);
  strut.style.height = (body.height + body.depth) + "em";
  strut.style.verticalAlign = -body.depth + "em";
  body.children.unshift(strut);

  return body;
}

/**
 * Take an entire parse tree, and build it into an appropriate set of HTML
 * nodes.
 */
export default function buildHTML(tree: AnyParseNode[], options: Options): DomSpan {
  // Strip off outer tag wrapper for processing below.
  let tag = null;
  if (tree.length === 1 && tree[0].type === "tag") {
    tag = tree[0].tag;
    tree = tree[0].body;
  }

  // Build the expression contained in the tree
  const expression = buildExpression(tree, options, true);

  const children = [];

```

```

// Create one base node for each chunk between potential line breaks.
// The TeXBook [p.173] says "A formula will be broken only after a
// relation symbol like $=$ or $<$ or $\rightarrow$, or after a binary
// operation symbol like $+$ or $-$ or $\times$, where the relation or
// binary operation is on the ``outer level" of the formula (i.e., not
// enclosed in {...} and not part of an \over construction)."

```

```

let parts = [];
for (let i = 0; i < expression.length; i++) {
  parts.push(expression[i]);
  if (expression[i].hasClass("mbin") ||
      expression[i].hasClass("mrel") ||
      expression[i].hasClass("allowbreak")) {
    // Put any post-operator glue on same line as operator.
    // Watch for \nobreak along the way, and stop at \newline.
    let nobreak = false;
    while (i < expression.length - 1 &&
           expression[i + 1].hasClass("mspace") &&
           !expression[i + 1].hasClass("newline")) {
      i++;
      parts.push(expression[i]);
      if (expression[i].hasClass("nobreak")) {
        nobreak = true;
      }
    }
    // Don't allow break if \nobreak among the post-operator glue.
    if (!nobreak) {
      children.push(buildHTMLUnbreakable(parts, options));
      parts = [];
    }
  } else if (expression[i].hasClass("newline")) {
    // Write the line except the newline

```

```

    parts.pop();
    if (parts.length > 0) {
        children.push(buildHTMLUnbreakable(parts, options));
        parts = [];
    }

    // Put the newline at the top level
    children.push(expression[i]);
}

}

if (parts.length > 0) {
    children.push(buildHTMLUnbreakable(parts, options));
}

// Now, if there was a tag, build it too and append it as a final child.
let tagChild;
if (tag) {
    tagChild = buildHTMLUnbreakable(
        buildExpression(tag, options, true)
    );
    tagChild.classes = ["tag"];
    children.push(tagChild);
}

const htmlNode = makeSpan(["parser-html"], children);
htmlNode.setAttribute("aria-hidden", "true");

// Adjust the strut of the tag to be the maximum height of all children
// (the height of the enclosing htmlNode) for proper vertical alignment.
if (tagChild) {
    const strut = tagChild.children[0];
    strut.style.height = (htmlNode.height + htmlNode.depth) + "em";
    strut.style.verticalAlign = (-htmlNode.depth) + "em";

```

```
}

```

```
return htmlNode;

```

```
}

```

parseTree.js

This is the parse tree method which is the backbone of the entire parser. It pares through the LaTeX file and creates a tree which is then converted into HTML node by node using the individual atomic functions.

```
import parser from "./Parser";

```

```
import ParseError from "./ParseError";

```

```
import type Settings from "./Settings";

```

```
import type {AnyParseNode} from "./parseNode";

```

```
/**

```

```
 * Parses an expression using a Parser, then returns the parsed result.

```

```
 */

```

```
const parseTree = function(toParse: string, settings: Settings): AnyParseNode[] {

```

```
  if (!(typeof toParse === 'string' || toParse instanceof String)) {

```

```
    throw new TypeError('Parser can only parse string typed expression');

```

```
  }

```

```
  const parser = new Parser(toParse, settings);

```

```
  // Blank out any \df@tag to avoid spurious "Duplicate \tag" errors

```

```
  delete parser.gullet.macros.current["\\df@tag"];

```

```
  let tree = parser.parse();

```

```
  // If the input used \tag, it will set the \df@tag macro to the tag.

```

```
  // In this case, we separately parse the tag and wrap the tree.

```

```
  if (parser.gullet.macros.get("\\df@tag")) {

```

```
    if (!settings.displayMode) {

```

```
      throw new ParseError("\\tag works only in display equations");

```

```
}  
parser.gullet.feed("\\df@tag");  
tree = [{  
  type: "tag",  
  mode: "text",  
  body: tree,  
  tag: parser.parse(),  
}];  
  
}  
  
return tree;  
};  
  
export default parseTree;
```

SubText Application Code

The main application implementing both function parsers and their function libraries and also providing a user interface along with a real-time database and login functionality and chatting and authentication services is the application - SubText. Built from the ground up on Angular on TypeScript.

It has been deployed on Google's firebase and can be seen [here](#). And also saves all messages and chats in real time on Google's Firebase real time database to provide a seamless user experience.

The project has been pushed on a [repository](#) on GitHub.

The application consists of 2 modules, one the main app module which provides all components and services and one a dedicated routing module.

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';
import { MessageComponent } from './message/message.component';
import { FormsModule } from "@angular/forms";
import { LoginComponent } from './login-page/login/login.component';
import { AppRoutingModuleModule } from './app-routing/app-routing.module';
import { DashboardComponent } from './dashboard/dashboard.component';
import { TestComponent } from './test/test.component';
import { AngularFireModule } from 'angularfire2';
import { environment } from '../environments/environment';
import { AngularFireDatabaseModule } from 'angularfire2/database'

@NgModule({
  declarations: [
    AppComponent,
    MessageComponent,
```

```

    LoginComponent,
    DashboardComponent,
    TestComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    AppRoutingModule,
    AngularFireModule.initializeApp(environment.firebase),
    AngularFireDatabaseModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

app-routing.module.ts

```

import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { LoginComponent } from "../login-page/login/login.component";
import { DashboardComponent } from "../dashboard/dashboard.component";
import { TestComponent } from "../test/test.component";

const routes: Routes = [
  { path: "", redirectTo: '/login', pathMatch: 'full'},
  { path: 'login', component: LoginComponent },
  { path: 'dashboard', component: DashboardComponent },
  { path: 'test', component: TestComponent }
];

@NgModule({

```

```
imports: [RouterModule.forRoot(routes)],
exports: [RouterModule]
})
```

```
export class AppRoutingModule { }
```

The app module consists of several services that each individually provide various real time functionality to the application.

chat.service.ts

The chat service connects to the firebase database and updates the database as new messages are sent and also updates the application as and when new messages are received.

```
import {Injectable} from '@angular/core';
import {Message} from '../message/message';
import {MessageType} from '../message/message-type.enum';
import {UserService} from './user.service';
import {AngularFirestore} from 'angularfire2/database';
```

```
@Injectable({
  providedIn: 'root'
})
```

```
export class ChatService {
```

```
  private mathematicsMessages: Message[] = [];
```

```
  private physicsGroupMessages: Message[] = [];
```

```
  private csGroupMessages: Message[] = [];
```

```
  private cs$;
```

```
  private mathematics$;
```

```
  private physics$;
```

```
  constructor(private readonly userService: UserService, private readonly database:
    AngularFireDatabase) {
```

```
    this.cs$ = database.list('/cs');
```



```

this.mathematics$ = database.list('/mathematics');
this.physics$ = database.list('/physics');
this.subscribeToDataChanges();
}

```

```

subscribeToDataChanges(): void {
    this.cs$.valueChanges().subscribe(messages => {
        this.csGroupMessages = messages;
    });
}

```

```

this.mathematics$.valueChanges().subscribe(messages => {
    this.mathematicsMessages = messages;
});

```

```

this.physics$.valueChanges().subscribe(messages => {
    this.physicsGroupMessages = messages;
});
}

```

```

getMessagesFor(groupName: string): Message[] {
    switch (groupName) {
        case 'maths': return this.mathematicsMessages;
        case 'physics': return this.physicsGroupMessages;
        case 'cs': return this.csGroupMessages;
        default: return [];
    }
}

```

```

sendMessage(message: string, group: string) {
    this.getMessagesGroup(group).push(
        new Message(this.userService.currentUser(), group, message,
this.getMessageTypeFromGroup(group))
    );
}

```

```

    );
}

getMessageTypeFromGroup(group: string): MessageType {
  switch (group) {
    case 'maths': return MessageType.LATEX;
    case 'physics': return MessageType.LATEX;
    case 'cs': return MessageType.MARKDOWN;
  }
}

```

```

getMessageGroup(group: string): Message[] {
  switch (group) {
    case 'maths': return this.mathematics$;
    case 'physics': return this.physics$;
    case 'cs': return this.cs$;
  }
}
}

```

clipboard.service.ts

This allows the user to copy any message she wants in the application to a global clipboard and this copied message can be normal text, emojis, markdown or even latex and on pasting this she will not get the preformatted text, but rather the underlying pre-formatted text used to generate the transpiled text so she can modify and edit and use pre-existing constructs.

```

import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class ClipboardService {

```

```

private static message = '';

constructor() { }

static set(message: string): void {
  this.message = message;
}

static getMessage(): string {
  return this.message;
}
}

```

The user can have only one message at a time on the clipboard.

group.service.ts

The group service is a very basic and atomic Immutable service class that allows the application to store the group that the user has currently selected.

```

import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class GroupService {

  currentlySelected: string;

  constructor() { }
}

```

markdown-parser.service.ts

This is a very important service class which ingests the `md-to-html-parser` library and calls the `makeHtml` function of the library using the message string we pass into the service.

```

import { Injectable } from '@angular/core';

```

```
// @ts-ignore
const showdown = require('md-to-html-parser');
const converter = new mdParser.Converter();
converter.setOption('parseImgDimensions', true);
converter.setOption('simplifiedAutoLink', true);
converter.setOption('strikethrough', true);
converter.setOption('tables', true);
converter.setFlavor('github');
```

```
@Injectable({
  providedIn: 'root'
})
export class MarkdownParserService {

  constructor() {}

  static parse(message: string): string {
    return converter.makeHtml(message);
  }
}
```

user.service.ts

The user service is also connected directly with the database and updates the database as when new users join the application and also removes users from the real-time database when users logout from the application. This service helps in maintaining a real-time base of who is using the application in real time.

```
import { Injectable } from '@angular/core';
import { AngularFireDatabase } from 'angularfire2/database';

@Injectable({
  providedIn: 'root'
})
```

```
export class UserService {

  private users = new Map<string, string>();
  private users$: Observable<string>;
  private activeUser: string;

  constructor(private readonly database: AngularFireDatabase) {
    this.users$ = database.list('/user');
    this.subscribeToChanges();
  }

  subscribeToChanges(): void {
    this.users$.snapshotChanges().subscribe(users => {
      this.users.clear();
      users.map(user => {
        this.users.set(user.payload.node._value_, user.key);
      });
    });
  }

  public currentUser(): string {
    return this.activeUser;
  }

  userNameExist(name: string): boolean {
    return this.users.has(name);
  }

  addUser(name: string) {
    this.users$.push(name);
    this.activeUser = name;
  }
}
```

```
logout(): void {
  this.database.object('/user/' + this.users.get(this.activeUser)).remove();
}
}
```

The application consists of 2 main components - the login component: that manages people logging in and logging out and the dashboard component that has various chat groups and messages being sent on these chat groups and also the message panel an real time message transpilation panel.

login.component.html

```
<div id="container">
  <div id="login-box">
    <div id="user-id-text">Enter Unique User Name</div>
    <label>
      <input type="text" [(ngModel)]="userName">
    </label>
    <div *ngIf="userNameExists()" id="invalid-username">The username has already been taken</div>
    <div id="join-button" (click)="joinChatRoom()" *ngIf="!userNameExists()">Join Chat Room</div>
  </div>
</div>
```

login.component.css

```
#container {
  height: 100vh;
  width: 100vw;
  position: relative;
}

#login-box {
  position: absolute;
  top: 50%;
  left: 50%;
```

```
transform: translateX(-50%) translateY(-50%);
```

```
background-color: lightcyan;
```

```
border: 1px solid black;
```

```
border-radius: 5px;
```

```
height: 600px;
```

```
width: 400px;
```

```
padding: 20px;
```

```
text-align: center;
```

```
text-transform: uppercase;
```

```
}
```

```
#login-box input {
```

```
border: none;
```

```
border-bottom: 1px solid black;
```

```
background-color: transparent;
```

```
max-width: 80%;
```

```
min-width: 300px;
```

```
margin-top: 100px;
```

```
outline: none;
```

```
}
```

```
#user-id-text {
```

```
margin-top: 100px;
```

```
font-size: 40px;
```

```
}
```

```
#invalid-username {
```

```
font-size: 10px;
```

```
color: darkred;
```

```
}
```

```
#join-button {  
  background-color: blueviolet;  
  border-radius: 5px;  
  color: white;  
  width: 200px;  
  margin-top: 50px;  
  padding: 10px;  
  
  position: absolute;  
  left: 100px;  
}
```

```
#join-button:hover {  
  cursor: pointer;  
  background-color: blue;  
}
```

```
#join-button.disabled:hover {  
  cursor: auto;  
  background-color: darkcyan;  
}
```

login.component.ts

```
import { Component, OnInit } from '@angular/core';  
import { UserService } from '../services/user.service';  
import { Router } from '@angular/router';  
  
@Component({  
  selector: 'app-login',  
  templateUrl: './login.component.html',  
  styleUrls: ['./login.component.css']  
})
```



```

})

export class LoginComponent implements OnInit {

  userName = '';

  constructor(private readonly userService: UserService,
               private readonly router: Router) { }

  ngOnInit() {

  }

  userNameExists(): boolean {

    return this.userService.userNameExist(this.userName);

  }

  joinChatRoom() {

    this.userService.addUser(this.userName);

    this.router.navigate(['dashboard']);

  }

}

```

The other component in the application is the main dashboard component.

dashboard.component.html

```

<div id="subtext">
  <div id="navbar">
    <span>Subtext &nbsp;  </span>
    <span id="currentUserTag">~{{userService.currentUser()}}</span>
    <div id="logout-button" (click)="logout()">Logout</div>
  </div>

  <div id="appPlaceHolder">
    <div id="groupsPane">
      <span>Groups</span>
    </div>
  </div>

```

```

<a><div class="group-card" (click)="updateSelectedGroup('maths')" [ngClass]="{
  'selected': groupService.currentlySelected === 'maths'
}"
>Mathematics</div></a>

```

```

<a><div class="group-card" (click)="updateSelectedGroup('physics')" [ngClass]="{
  'selected': groupService.currentlySelected === 'physics'
}"
>Physics</div></a>

```

```

<a><div class="group-card" (click)="updateSelectedGroup('cs')" [ngClass]="{
  'selected': groupService.currentlySelected === 'cs'
}"
>Computer Science</div></a>
</div>

```

```

<div id="messagesPane">
  <div id="messagesContainer">
    <app-message *ngFor="let message of
chatService.getMessagesFor(groupService.currentlySelected)"
      [sender]="message.sender" [timestamp]="message.timestamp"
      [utf8Message]="message.utfContent"
      [ngStyle]="{'align-self': message.sender === userService.currentUser() ? 'flex-end' :
'flex-start'}">
      <span *ngIf="message.messageType === 0" [innerHTML]="message.innerHtml"></span>
      <ng-katex-paragraph *ngIf="message.messageType === 1"
[paragraph]="message.utfContent"></ng-katex-paragraph>
    </app-message>
    <span *ngIf="chatService.getMessagesFor(groupService.currentlySelected).length === 0"
style="align-self: center">No Conversations here yet</span>
  </div>

```

```

<div id="textInputsContainer">

```

```

<div id="textareas">
  <textarea placeholder="Enter Message Here" [(ngModel)]="message"></textarea>
  <div id="compiledTextMessage">
    <div>
      <span *ngIf="isMarkdownMessage()" [innerHTML]="getMarkdownMessage()"></span>
      <ng-latex-paragraph *ngIf="isLatexMessage()"
[paragraph]="message"></ng-latex-paragraph>
    </div>

    <i class="far fa-clipboard" style="font-size: 24px" (click)="copyMessageFromClipboard()"></i>
  </div>
</div>

<button (click)="sendMessage()">SEND</button>
</div>
</div>

</div>
</div>

```

dashboard.component.css

```

#subtext {
  display: grid;
  grid-template-rows: 55px 1fr;
  height: 100vh;
  width: 100vw;
}

#navbar {
  display: grid;
  grid-template-columns: auto 1fr auto;
  padding-left: 20px;
  box-shadow: 0 0 5px 0;
}

```

```
align-items: center;
}

#navbar > span {
  font-size: 2em;
}

#logout-button {
  background-color: blueviolet;
  color: white;
  padding: 15px;
  border-radius: 5px;
  margin-right: 20px;
}

#logout-button:hover {
  cursor: pointer;
}

#appPlaceholder {
  display: grid;
  grid-template-columns: 350px 1fr;
  max-height: 100vh;
}

#groupsPane {
  display: flex;
  flex-direction: column;
  box-shadow: 0 10px 0 0;
  border-right: 1px solid grey;
}

#groupsPane > span {
  font-size: 2.3em;
```

```
padding: 10px;
}
```

```
#groupsPane > a {
text-decoration: none;
}
```

```
.group-card {
padding: 10px;
font-size: 1.5em;
border: .1px solid grey;
}
.group-card:hover {
background-color: lightgray;
}
.group-card.selected {
background-color: cyan;
}
```

```
#messagesPane {
display: grid;
grid-template-rows: 1fr 150px;
/*overflow: hidden;*/
/*max-height: 80vh;*/
}
```

```
#messagesContainer {
/*background-color: grey;*/
display: flex;
flex-direction: column;
padding: 10px;
max-height: 80vh;
```

```
/*max-width: 50vw;*/
overflow: scroll;
}

app-message {
margin: 10px 0;
/*align-self: flex-end;*/
}

.align-self-start {
align-self: flex-start;
}

.align-self-end {
align-self: flex-end;
}

#textInputContainer {
display: grid;
grid-template-columns: 1fr 80px;
}

#textInputContainer > button {
background-color: lightblue;
border: none;
font-size: larger;
}

#textareas {
display: grid;
grid-template-rows: 1fr 1fr;
}

#textareas > textarea {
resize: none;
```

```
padding: 0;
margin: 0;
}
```

```
#compiledTextMessage {
display: grid;
grid-template-columns: 1fr 10px;
/*justify-items: center;*/
align-items: center;
padding-right: 20px;
}
```

```
#compiledTextMessage > i: hover {
cursor: pointer;
color: darkviolet;
}
```

dashboard.component.ts

```
import { Component } from '@angular/core';
import { GroupService } from '../services/group.service';
import { ChatService } from '../services/chat.service';
import { UserService } from '../services/user.service';
import { MarkdownParserService } from '../services/markdown-parser.service';
import { ClipboardService } from '../services/clipboard.service';
import { Router } from '@angular/router';

@Component({
  selector: 'app-dashboard',
  templateUrl: './dashboard.component.html',
  styleUrls: ['./dashboard.component.css']
})
export class DashboardComponent {
```

```
title = 'subtext';
message = "";

constructor(readonly groupService: GroupService,
             readonly chatService: ChatService,
             readonly userService: UserService,
             readonly router: Router) {
}

updateSelectedGroup(groupName: string) {
    this.groupService.currentlySelected = groupName;
}

isLatexMessage(): boolean {
    return this.groupService.currentlySelected === 'maths' || this.groupService.currentlySelected === 'physics';
}

isMarkdownMessage(): boolean {
    return this.groupService.currentlySelected === 'cs';
}

getMarkdownMessage(): string {
    return MarkdownParserService.parse(this.message);
}

sendMessage() {
    this.chatService.sendNewMessage(this.message, this.groupService.currentlySelected);
    this.message = "";
}

copyMessageFromClipboard(): void {
```



```

    this.message += ClipboardService.getMessage();
}

```

```

logout(): void {
    this.userService.logout();
    this.router.navigate(['login']);
}
}

```

A helper component being used in the application is the message component. It is the small component being sent in the chatting application between users in every group.

message.component.html

```

<div id="messageBox">
  <div id="sender">{{sender}}</div>
  <div id="timestamp">{{timestamp}}</div>
  <div id="content-container">
    <ng-content></ng-content>
    <i style="font-size:20px" class="far fa-copy" (click)="copyToClipboard()"></i>
  </div>
</div>

```

message.component.css

```

#messageBox {
  background-color: lightcyan;
  border-radius: 5px;
  padding: 10px;
  width: 300px;
}

#messageBox > #sender {
  font-size: 1.4em;
  text-transform: lowercase;
}

```

```
#messageBox > #timestamp {  
  font-size: .6em;  
  margin-bottom: 10px;  
}
```

```
#messageBox > #content-container {  
  display: grid;  
  grid-template-columns: 1fr 10px;  
  grid-gap: 10px;  
  padding-right: 10px;  
}
```

```
i:hover {  
  cursor: pointer;  
  color: darkviolet;  
}
```

message.component.ts

```
import {Component, Input, OnInit} from '@angular/core';  
import {ClipboardService} from '../services/clipboard.service';
```

```
@Component({  
  selector: 'app-message',  
  templateUrl: './message.component.html',  
  styleUrls: ['./message.component.css']  
})
```

```
export class MessageComponent implements OnInit {  
  @Input() sender: string;  
  @Input() timestamp: Date;  
  @Input() utf8Message: string;  
  
  constructor() { }
```

```
ngOnInit() {  
}  
  
copyToClipboard(): void {  
  ClipboardService.set(this.utf8Message);  
}  
}
```

Along with all the components, services and modules in the application we also have 2 models that are being used to transmit data within the application and with the server.

message.ts

```
import {MessageType} from './message-type.enum';  
import {MarkdownParserService} from '../services/markdown-parser.service';  
  
export class Message {  
  sender: string;  
  group: string;  
  timestamp: string;  
  utfContent: string;  
  innerHtml: string;  
  messageType: MessageType;  
  
  constructor(sender, group, utfContent, messageType: MessageType) {  
    this.sender = sender;  
    this.group = group;  
    this.timestamp = new Date().toString();  
    this.utfContent = utfContent;  
    this.innerHtml = this.getTranspiled(utfContent, messageType);  
    this.messageType = messageType;  
  }  
}
```

```
getTranspiled(content: string, messageType: MessageType): string {  
  switch(messageType) {  
    case MessageType.LATEX: return content;  
    case MessageType.MARKDOWN: return this.parseMarkdownMessage(content);  
  }  
}
```

```
parseMarkdownMessage(content: string): string {  
  return MarkdownParserService.parse(content);  
}  
}
```

message-type.enum.ts

```
export enum MessageType {  
  MARKDOWN,  
  LATEX  
}
```

Conclusion

We have seen that with subtext users and organizations can now send messages that contain more contextual data and just a better representation of scientific, mathematical and even preformatted based data.

We have seen that by adding markdown support in a normal communication real-time chatting application we are adding a very friendly feature for developers and programmers who can now share code snippets not as pictures just to get some formatting, or as plain text so that the other person has to copy and paste, but rather both programmers will now be able to see the highlighted code with all its colors and in all its glory right in the messaging pane which really improves their experience and gives them incentive to jump ship from slack and whatsapp and messenger or IRC to one mainstream application.

We have also seen that the application further adds LaTeX compilation and transpilation with an excellent formatting and this adds further incentive for physicists and mathematicians to come to a common platform where they can share in the context they are familiar with and also use normal chatting features.


We have even seen that the transpilation of the LaTeX in the application to the HTML using a syntactic parse tree is based on a highly efficient algorithm that has been derived from the Context Free Grammar in Theory of Computation and the parsing to create the parse tree is being done using a regular language based on a Finite State Automata that distinctly maps out all symbols/tokens that one can encounter in LaTeX and creates the parse tree in the most time efficient manner.

Furthermore all calculations and transpilations are taking place on the client side and not on the server side. This greatly reduces any load that would've otherwise fallen on the server and also improves the user experience as now the users can type out messages and see the output instantaneously without even requiring the internet.

So, it can be concluded here that the addition of markdown and LaTeX that provide instantaneous markup and client side transpilation that takes place without any extra resources or the internet adds makes this application a standout application and with further work and funding this project can become a viable and lucrative alternative to the plethora of messaging applications that we have these days.

Bibliography

1. [Firebase Realtime Database](#) [Firebase-Google]
2. [Firebase Hosting](#) [Firebase-Google]
3. [Firebase CLI](#) [Firebase-Google]
4. [typescript-lang](#) [Microsoft]
5. [Repository Link](#) [github]
6. [Webstorm](#) [Jetbrains]
7. [Datagrip](#) [Jetbrains]
8. [Latex-symbols-sheet](#)
9. [Markdown Features Sheet](#)
10. [Overleaf](#)
11. [Whatsapp Messenger](#) [Wikipedia]
12. [Facebook Messenger](#) [Wikipedia]
13. [Slack](#) [Wikipedia]
14. [Git](#)
15. [GitHub](#)
16. [Angular](#) [Google]
17. [Node](#)
18. [Npm](#)
19. [Introduction to Theory of Computing](#) [MIT Press]
20. [Angular Hosting on Github Pages](#) [Telerik]
21. [gh-pages](#) [npm]
22. [Introduction to Theory of Computation](#) by Michael Sipser [Cengage Learning]
23. [Deterministic Finite State Automaton](#) [Wikipedia]
24. [Clean Code](#) by Robert C. Martin (Uncle Bob) [Amazon]
25. [GraphViz](#)
26. [DotScript](#) [Wikipedia]
27. [Introduction to Lexical Analysis](#) [Geeks for Geeks]
28. [Lexical Analysis](#) [Wikipedia]
29. [Parse Tree](#) [Wikipedia]
30. [Abstract Syntax Tree](#) [Wikipedia]
31. [Parsing](#) [Wikipedia]
32. [Structure and Interpretation of Computer Programs](#) by Harold Abelson and Gerald Jay Sussman with Julie Susman [Wikipedia]
33. [Structure and Interpretation of Computer Programs](#) by Harold Abelson and Gerald Jay Sussman with Julie Susman [mit press]
34. [Context Free Grammar](#) [brilliant-org]
35. [Latex-project](#)

- 
36. [Introduction to LaTeX](#) [latex-project]
 37. [LaTeX](#) [wikipedia]
 38. [Online Introduction to LaTeX](#) [Overleaf]
 39. [Getting started with Tex, LaTeX and Friends](#) [tug-org]
 40. [LaTeX Cheat sheet](#) [github]
 41. [A short introduction to LaTeX2e](#) [CTAN]
 42. [md-to-html-parser](#) [GitHub]
 43. [latex-to-html-parser](#) [GitHub]