University of Twente

Design Project - Technical Computer Science

LaTeXt To Speech

Design Report

Author:
Joris Bosman *(s2625768)*
Beitske Flake *(s1592335)*
Xander Heij *(s2571250)*
Erik Oosting *(s2105136)*
Jeroen Zwiers *(s2319187)*

Supervisor:
Dr. Ir. Mahboobeh Zangiabady

April 24, 2023

UNIVERSITY OF TWENTE.

Abstract

In this report, we will discuss the process and results of our work on the "Design Project", one of the final courses of the Bachelor Technical Computer Science at the University of Twente. During this course, we as a group looked into the design and development of a system that converts the contents of LaTeX files into audible speech. Our goal here was to develop a system that not only takes the verbatim textual contents but includes logical contextual pronunciation of the possible maths, mathematical formulas and programming code listings in the final audio as well. By creating such a system, we hope to support students with visual or learning disabilities in their academic and educational efforts and people that prefer to read or study papers by listening to them.

We were supervised during the Design Project by our supervisor and client dr. ir. M. Zangiabady, whom we would like to thank for all her help, enthusiasm and guidance during this time.

Contents

Chapter 1

Introduction

In this chapter, our project and approach will be introduced. Furthermore, we will give a short overview of what to expect in this report.

1.1 Problem statement & Motivation

For students (or academic employees) with visual or learning disabilities, it can be a challenge to read scientific papers and other documents necessary for their studies or projects. Nowadays, these papers (especially in so-called hard sciences) often come in the form of LATEX files [1] that get converted to PDF or other reading formats. Realising that reading scientific papers and other educational contents is crucial for all kinds of academic education and work, it would be beneficial to develop a system that provides the contents of the aforementioned documents in an audible and therefore non-visual format.

Doing research on how to design and develop such a system, and how to improve this system in comparison to already existing solutions, can greatly benefit students with visual and learning disabilities. And because our aim for this system is to be used for educational and academic purposes, we want to include academic elements in these files, such as pseudo-code listings.

1.2 Objectives

To solve the problem, a couple of objectives are determined. Most importantly, the text of the resulting PDF-like file must be spoken aloud to the user. This is an overarching objective, as it includes two sub-goals, of which at least one will be implemented. Firstly, all mathematical formulas should be read in a well-structured manner. Secondly, pseudo-code should also be read to the user in a way that makes sense.

A different objective is for the application to be visually designed and built for users with a visual disability to improve accessibility to its maximum. Finally, one of our goals was to have fun designing, building, testing, and writing the report.

1.3 Approach

The first phase of the project is the research phase, where we research everything we can about the problem statement, which is comprised of research on the intended user group, the LATEX language, and speech synthesizers.

The second phase of the project is the design phase, where we think about the problem statement and come up with solutions. This phase includes multiple parts. To start, we think about the main structure of the application. Then, we design the graphical user interface of the application, making sure we incorporate the research previously done about the intended users in the design.

The third phase of the project is the testing and implementation phase. The proposed design will be implemented and improved upon after testing.

The final phase of the project is the finalization phase, including presenting our system and results.

1.4 Structure of Report

In the first part of this report, the project and its background will be described. Second, we will look at requirement engineering and specification. After this, the global and detailed design of our system will be explained, including important design choices with motivation. In the Implementation and Testing the System chapters we will explain more about the code and frameworks of our system, and show the system was tested. The chapter Future Planning will explain more about what will happen to the project and system in the future, and how it can be expanded on. Finally, there will be a discussion and conclusion chapter on our project and its results.

Chapter 2

Research & Project Analysis

2.1 Problem Background

In chapter 1 we explained the aim to develop a system where a user can listen to a compiled LaTeX file. The client explained that she and a group of students worked on the problem before, but the other way around: They developed a system where speech was converted to (LaTeX) files. The goal for our system remains the same, namely to help students and employees with their academic efforts.

The client made clear that she wants a system that not only could convert normal LaTeX texts to speech, but also made the mathematics or code listings pronounceable, and preferable both in combination with normal 'plain' text. There are already Text To Speech (TTS) systems available (see also section 2.3 below), but our goal was to include the aforementioned difficult parts and build a proper system around it taking our target audience into account.

Because of possible existing software and libraries that could be used for the system, and because of the nature of the target audience as explained in the next section, we decided to do more research into existing TTS systems and disability-friendly applications first.

2.2 Target Audience & Future Users

In this section, we will analyse the target audience of our system and their needs, based on our client's assignment. The future users of our system consist of two groups:

1. Users with visual and/ or learning disabilities.
2. Users with a preference for listening to academic materials or papers.

The first target group can of course also be part of the second target group.

2.2.1 Visual & Learning Disabilities

The first group contains a wide variety of possible disabilities. Learning disabilities are described as "a diverse group of disorders in which children who generally possess at least average intelligence have problems processing information or generating output" [2]. A well-known example within learning disabilities is dyslexia, with Handler et al. describing how many (young) adults with dyslexia still suffer from lower reading rates when reading at a later age, even when treated as a child, and "gradually increasing complaints of discomfort, eyestrain, headache, blurred vision, or diplopia during extended periods of studying". Providing a method of listening to materials instead of reading them can therefore be very beneficial.

Aside from learning disabilities, there is a wide variety of visual disabilities including but not limited to vision impairments and (partial) blindness [3]. Some of these vision impairments of course go hand in hand with learning disabilities. It is clear why visual disabilities would prevent a person from reading a paper. Brzoza et al. already state how recent development in more affordable computers allow more visually impaired people to make use of methods like text-to-speech to read books [4], so it would make sense to provide this availability for LaTeX and academic files as well.

2.2.2 Audio Preferences For Academic Materials

Furthermore, there is the audience that simply prefers to listen to academic content rather than read it. For example, one needs to read an important paper for their research but has little time to do so. However, there is an audio reading of said paper available, and they listen to the paper in the car or train while commuting from work to home, thus still processing the contents of the paper.

2.3 Existing Solutions & Systems

In our research, we found several systems worth mentioning. First of all, TTS applications are already available, but often as a browser extension or downloadable app or integration. A known example of this is Speechify[5]. However, these systems often focus on simply reading text and not necessarily files or even LaTeX files. Besides, these systems often charge for premium features or for reading more than X words.

So what is there to find in terms of LaTeX to Speech? We encountered the free Apache licensed program 'MathJax LaTeX to Speech Converter' [6], but this was mainly math focused and with a very robotic voice. There were more interesting sources and solutions, with Tex2Speech [7] as a notable one. Unfortunately, this one required an AWS (Amazon) account to be used and did not make use of a real LaTeX processor according to the documentation. This means that it would not support a lot of necessary LaTeX functionality which we want to support. However, it did provide insight into how to build a Tex parser and generate speech out of it.

2.4 Conclusion

In conclusion, it is clear why the target audiences would prefer listening to papers above reading them, and why it would even be beneficial to them. There are some existing solutions, but not all of them are free, and there are improvements that could be made to them. These improvements include but are not limited to providing proper math or code pronunciation and different selections of voices.

Chapter 3

Requirement Engineering

3.1 Requirement Specification

To determine the requirements for our system, we spoke with the client and did research into accessible programs for both learning and visual disabilities. The process of formulating these requirements was led by Agile practices as explained below.

3.1.1 Agile Project Management Practices

We chose to use Agile (software) Project Management practices [8] for both our project management and requirement specifications. This was done to keep a proper structure in the planning and development of our entire project, from requirement engineering to testing.

Following the Agile approach, the formulated requirements were divided and planned into sprints with a duration of one week. This was based on the SCRUM methodology [9], which we also decided to use in our project to improve collaboration within our group. This also resulted in the use of the daily scrum or daily stand-up meetings to keep everyone informed of the daily progress. Finally, we made use of a Trello board [10] to keep track of both the requirements and the sprints.

3.1.2 Requirement Formulation Methods

The requirements were based on the information and priorities provided by the client and other stakeholders, such as the future users of our systems. The MoSCoW [11] method was chosen to prioritize the requirements and determine critical functionality.

3.2 Requirement Analysis

The next two sections show the user and system requirements which were formulated together with the client and other stakeholders.

3.2.1 User requirements

As a user, I want to:

1. select and upload a LaTeX document.
2. play/pause the playback of the speech.
3. click on a segment to start playback from there.
4. type LaTeX in a text editor.
5. change the colors in the GUI.
6. change the font or font size in the GUI.

7. choose a voice from a set of voices.
8. download the speech as an audio file.
9. select other languages.
10. navigate the interface with keyboard shortcuts.

3.2.2 System requirements

The system should:

1. verify that the LaTeX is valid.
2. be able to speak the processed LaTeX.
3. be able to extract math formulas and convert them to speech.
4. be able to extract (python) code and convert this to speech.
5. divided the LaTeX in segments and display them in a preview
6. highlight the segment that is currently being spoken.
7. automatically scroll through the preview to the current segment.
8. highlight compile errors in the text area.
9. have syntax highlighting in the text area.
10. be able to help the user by providing programming tools in the editor.
11. save the text and voice in cache so it is not lost on reload.
12. provide a help section to guide the user.
13. have code documentation for future clarity and maintenance.

Chapter 4

Global Design

4.1 Proposed Global System Design

4.1.1 General Idea

The overall goal of the application is to have an editor with Text-to-speech capabilities built in. We decided to base our layout on the latex editor Overleaf, as will give a sense of familiarity with people who have used the LaTeX editor before.

4.1.2 Software, Programming Languages & Frameworks

The application is written in `C#` and `JavaScript`, and utilizes a framework called Blazor [12]. Blazor is a framework that allows us to write a web application using `C#` that is compiled to WebAssembly [13] and can be executed directly in the browser. We decided it is important for our project to be a web application because it can then be used in most browsers, making it accessible and cross-platform. The reason we chose Blazor is so we can write the LaTeX parser in `C#`, rather than `JavaScript`. This is an important decision because `C#` is a much faster language and thus has a big positive impact on the performance of our parser. On top of that, there is the added benefit of `C#` having much better type safety, and overall being a much better programming language to be working within a larger project.

4.1.3 System Mock-ups

The Lo-Fi prototypes, mock-ups and initial interface design were done in Figma and were used for usability testing. The mock-ups can be found in appendix A.

Our first design took a lot of inspiration from Overleaf. Inspiration on color, styling and fonts were also drawn from a dyslexia style guide [14] provided by the British Dyslexia Association and a paper by Rello & Bigham [15] on background colors for readers with and without visual disability. This is why we took light, but mostly non-white colors for our backgrounds (though we did take white to emulate a paper background for the first designs). The usage of red and green colors was also avoided, as well as italic fonts. The original design also displays a large bar of blue on the left. This could be potential space to handle a file tree in case we wanted to handle a LaTeX project with multiple files, as mentioned in 8.2.

We initially created 2 prototypes, one where the media buttons were spread out across the bottom (fig A.1), and one where it just took up the left column (fig A.2). Both versions were brought in an iterated version to our supervisor/client to gain feedback on her user experience and preference.

Before the aforementioned meeting, there was the decision to iterate on the design of figure A.1, and create some improvements (fig A.3). Most notably, we had decided were not going to implement a multiple files system. With this change in mind, the buttons were moved from the top to the empty space on the side. This created bigger, easier-to-click buttons. Later on, we added missing (additional) features to our design in figure A.4, such as a button to compile the source LaTeX code and a voice selection dropdown.

4.2 System Overview

The sections below give a global overview and a short description of the most important parts of the system.

4.2.1 LaTeX Editor

The initial plan was to simply have a file upload. But we decided it is useful to have a text editor to quickly and easily change the LaTeX without having to re-upload a file. This text editor will have most basic features you would expect from a text editor: line numbers, basic LaTeX syntax highlighting, and automatic matching of brackets.

4.2.2 LaTeX Preview

The LaTeX preview will display the compiled LaTeX code. This preview will be generated in `HTML`, and it will be divided into segments. Dividing the preview into segments will allow us to highlight the segment currently being spoken, and we will be able to make each segment clickable, so the user can skip playback to a specific segment.

4.2.3 Audio Player

To control the playback of the speech, we will add basic audio controls. This includes a play/pause button, buttons to skip forwards and backwards through the segments in the preview, a slider to control the rate of speech, and a toggle to enable automatic playback. Automatic playback entails automatically moving on to the next segment, as opposed to stopping playback after each segment.

4.2.4 Help Page

To explain all the features of the user interface we will provide a help page. This page will function just like the LaTeX preview, it can be spoken aloud, and it will highlight the segment that is currently being spoken. The page will also provide an overview of the hotkeys that can be used to navigate the system.

4.2.5 Settings Page

Because it was established that customizability is important when trying to style a page for different visual needs, we will provide an options page. This page will have options to change the color of every styled part of the system, but also to change the fonts and font size.

Chapter 5

Detailed Design

5.1 Modeling the System

Based on our global design and the gathered requirements, we created several UML diagrams to explore the design and inner workings of our system better. This was not only a good way to visualize our initial designs, but to check our requirements against our proposed system as well.

5.1.1 User Interaction With The System

A very important part of the design process was determining how users would interact with our system. We were already making interface style choices based on our research but needed an overview of how our users would interact with our system considering their use cases as well. Therefore it was decided to model our most important data overviews and interactions, as well as the goals and validity of our requirements, in different types of UML models as described by Pohl et al.[16].

Use Case Diagram

The Use Case Diagram from figure 5.1 models the interaction between the user and the system. The system itself exists of 2 actors: the LaTeX parser, and the Browser TTS extension. The user is the 'student'.

In the first iterations of this diagram, we made some implicit design decisions, which don't apply anymore and got changed in the next iterations. For instance, we initially referred to the user as 'student', which is too specific considering our broader target audience. A more suitable and inclusive name would be simply 'user'.

Furthermore, we differentiate between the use case of uploading a file and uploading text in this version of the diagram. The text upload use case means that a user wants to paste text directly into the editor. This design decision was implemented, but the option to drag and drop a file was also added later on.

The parser will have the latex as input, and the parsed text as output. We modelled the system in such a way that we would make the parser so that the parsed text could be both used to retrieve the audio and to display the parsed text as HTML in the preview. The different segments can be highlighted as well, although this was still an optional additional use case in the first iterations. The user can play the audio and has the option to control the playback, by going to the next or previous segment and by using the play/pause button. There is no relation in the use case diagram between the use case of playing audio and retrieving audio, since the audio should be processed beforehand such that it is always available if the user requests it, without long delays.

5.1.2 Process flow Diagram

The data flow diagram in figure 5.2 is made to model how the different system actors (also partly mentioned in the use case diagram) will work, and how their input and output connect to the rest of

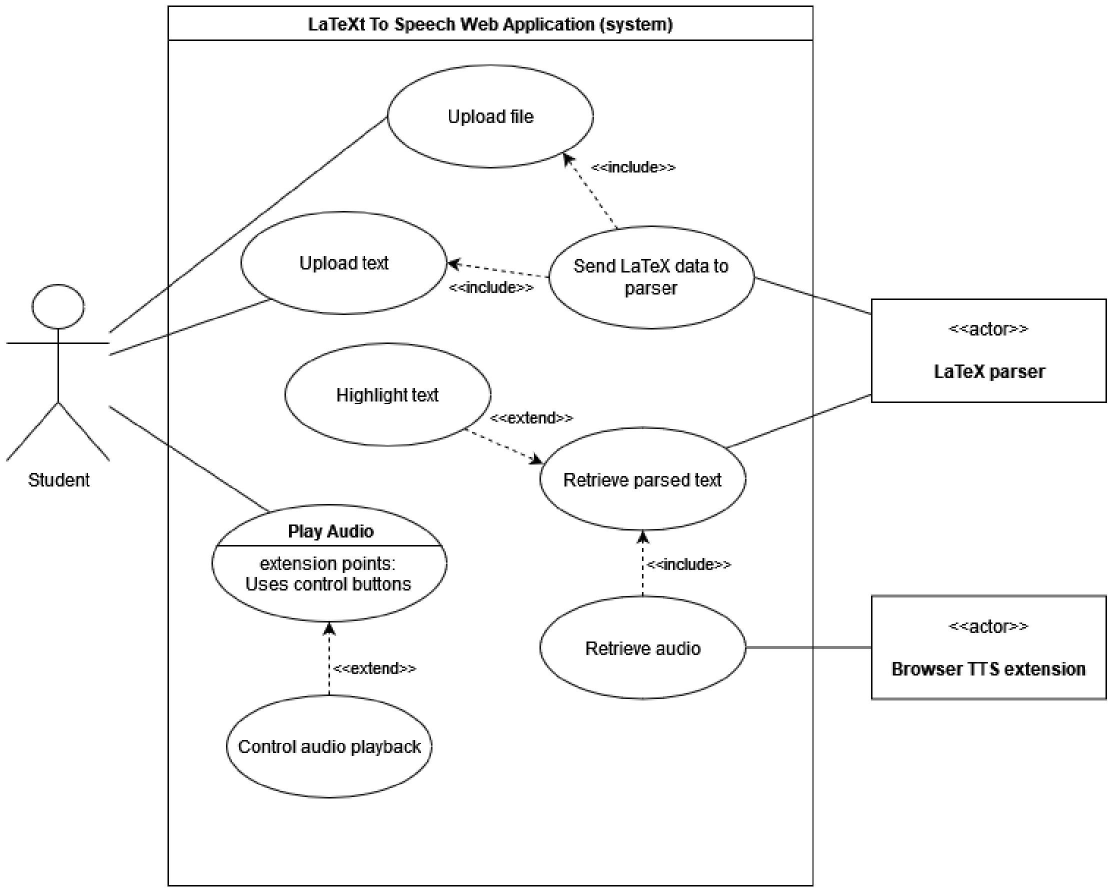


Figure 5.1: Use Case diagram of the interaction between the user and the LaTeXt to Speech system

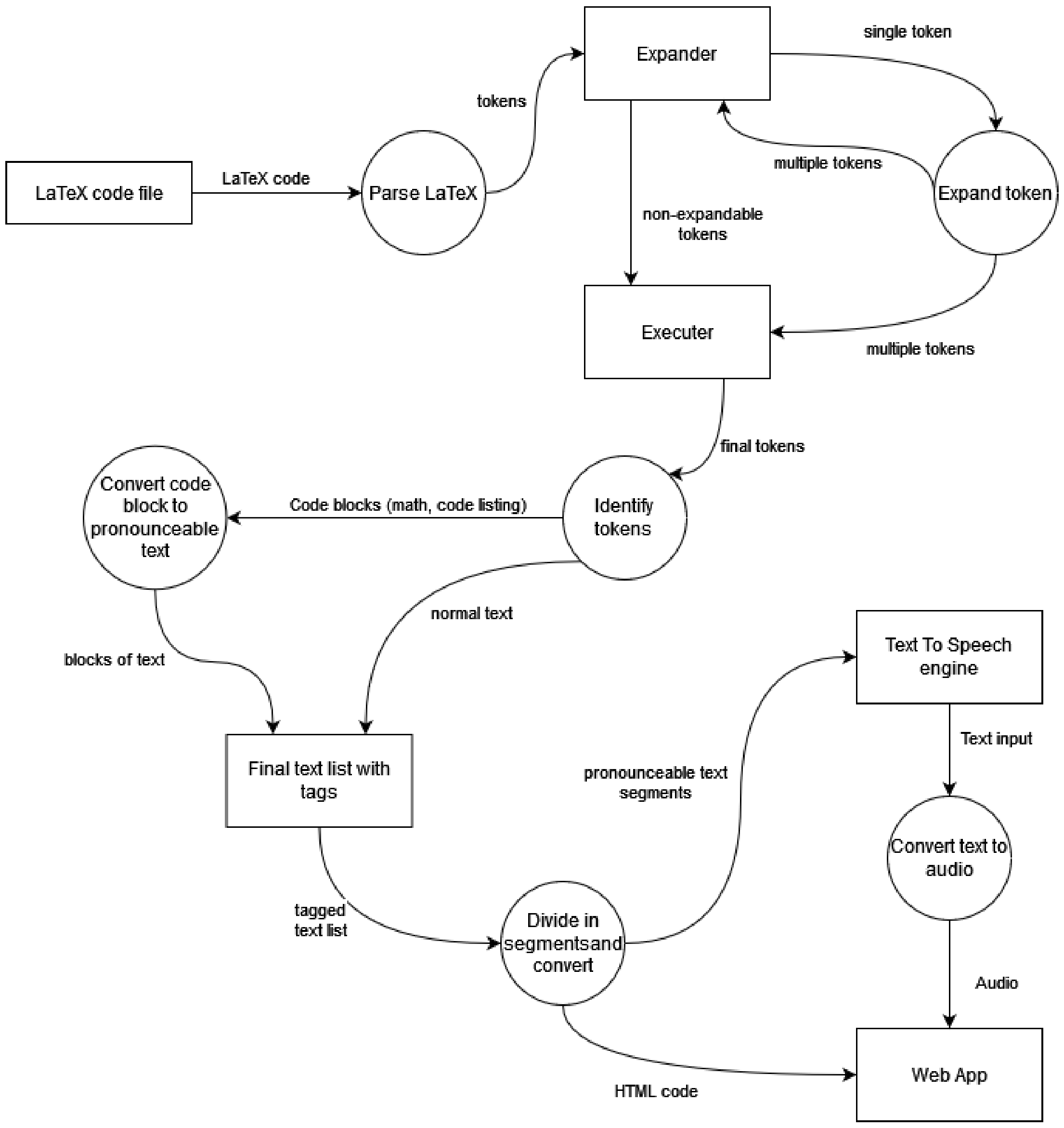


Figure 5.2: Process flow diagram

the application. It had already been decided that we would not make a text-to-speech engine ourselves, so this has been presented by a 'Text to Speech engine' block with data input and output.

As we did decide to make the parser ourselves, the data flow in the parser is modelled in more detail to gain a good understanding of the data flows in our LaTeX to speech/text infrastructure. Based on extensive research on TeX's and LaTeX's inner workings, we could already model the parser and its data flow in a way which is very close to the actual implementation. We decided to differentiate between math, code and normal text because these will need to be processed differently when converting to speech.

The diagram starts with LaTeX code, which is first ran through our parser. Exactly how the parser works is explained in detail in section 6.2.1. The final step in the parser is the Executor, which results in a list of tokens. This list is then divided into an identified list of math, code, and normal text parts, which are then further divided into shorter segments. These segments are then both converted to `HTML` code and sent to the TTS engine. Finally, the `HTML` is displayed in the web app, and the resulting audio from the TTS engine is linked to those `HTML` segments.

The final implementation of the parser was kept very close to the shown data flow diagram.

Activity diagram

The activity diagram in figure 5.3 models the interaction between the user and the application for the 'playing audio' activity. The focus of this diagram lies on the interaction with the front end, so the back end isn't modelled in much detail. For more details on the back-end, we refer to the flow diagram in figure 5.2 and section 6.2.1.

The modelled interaction starts with the upload of a file. In the diagram, it has to be checked for validity first, and then it gets parsed. However, we ended up implementing this in a different way: The file upload only allows to upload `.tex` or `.txt` files. The parser can handle other files in theory, so it shouldn't give an error when you would parse a file of a different type. Therefore we also don't show an error message for this case anymore, but we still do give error messages for parsing errors in the LaTeX file itself.

A design decision made in this diagram is that the audio does not automatically starts playing. The user has to click on the start button or a segment first. Something we did not consider yet in this diagram was to have autoplay as an option that can be turned off. The diagram assumes that autoplay is on for clicking on the play button, and off for clicking on a segment. We did decide on implementing the autoplay that could be interacted with by the user in a later version because we thought it would give the user more control.

The remainder of the diagram is close to the actual implementation. It is possible to play and pause audio at any time, as well as select another segment to play or upload another file.

5.2 Design Choices

5.2.1 LaTeX Parser

We made the decision to create our own LaTeX parser instead of relying on existing ones. This gives us more control over the output. This makes it easier to extract all the text that has to be spoken and also helps extract the math and code segments, which we have to process separately to properly convert those to pronounceable text. On top of that, we want to have a preview of the compiled LaTeX, and although parsers that convert LaTeX to HTML exist, we want to do segment highlighting when a segment is being spoken. This is made easier by having our own parser. Another minor reason is the text editor, although likely also possible with existing parsers, implementing our own parser will make it easier to incorporate things like error highlighting in the text area.

5.2.2 Mathematics and Code to Speech Conversion

We use a multitude of methods to convert certain parts of the document to speakable text. Which is then sent to a 3rd-party API, described in section 5.2.3.

While most text is handled by our own parser in a relatively straightforward manner, there are 2 special cases that we will describe below

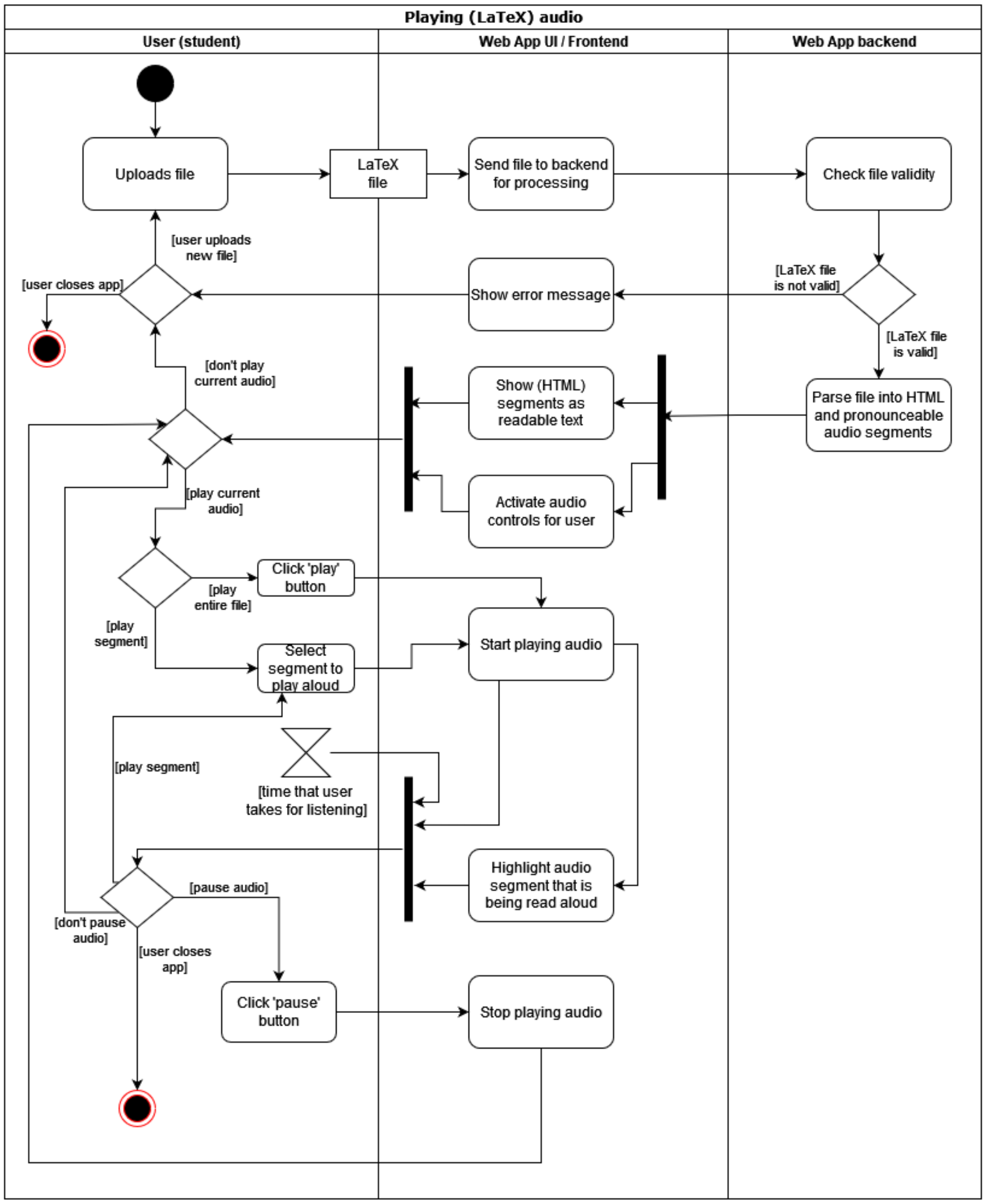


Figure 5.3: UML Activity diagram on the interaction between the user and the system when uploading and playing audio.

Math

To transform the mathematics to human text, we used MathLive [17]. MathLive parses LaTeX already, so we can just pass it the raw LaTeX.

Code

The rendering and TTS conversion of code blocks is relatively straightforward. We make sure that we maintain the indentation of the code in the source file when rendering it to HTML. In a code block, characters such as `(){}` are pronounced explicitly, because this is often necessary to understand the code. On top of that, there is another primitive parser to highlight identifiers and keywords in code blocks in the text editor.

5.2.3 Text-To-Speech Conversion Methods

We currently use the built-in browser for text-to-speech conversion. The in-browser text-to-speech mechanism is still kind of robotic sounding. We have tried looking for more natural-sounding alternatives, but all of the alternatives we considered only have paid options for proper-sounding voices [5], or do not have better voice options than the in-browser TTS converter provides.

5.2.4 System look and usability

The color theme of our website is completely customizable. As also mentioned in earlier chapters, this was done to ensure accessibility to all of our users. For the default color scheme, we created a scheme that was pleasing to the eye and used fonts that were easy to read for someone with learning disabilities such as dyslexia. Examples of those fonts are Arial, Verdana, Calibri and Century Gothic.[14] More disability-friendly style choices [15] for the default theme included the use of non (sharp) whites, dark text on light colours, the absence of distracting patterns, the absence of italic fonts and the avoidance of colors like green and red.

Chapter 6

Implementation

6.1 Implementation Deviations

6.1.1 Speechify

Originally we planned to use the speech engine that is built into all desktop browsers. However, due to the lack of different voices, multiple languages, and quality, we opted instead to look for an API. After searching for a potentially free API, we eventually found Speechify[5]. A text to speech engine that we thought had a free option. This was a great find, as this API had multiple languages with most of them having both a male and female voice. This meant a simple request to the API returned a nice audio fragment that we could make use of in the browser. See 6.2.2 for further explanation. Speechify had a copyright waiver that was not really clear. Therefore, we decided to not include Speechify in our project. Instead, the code now contains the boilerplate for an API implementation. This can then easily be changed to fit any API's needed information: 6.2.2.

6.2 Coding Details

6.2.1 LaTeX Parser

LaTeX is usually parsed in 4 steps: input processing, expanding, executing, and finally rendering. For our project, we only need the first 3 steps, as the final rendering step is done separately as HTML. These steps are heavily intertwined, but it is easier to think of them as separate steps when explaining them. The input processing is the easiest of them all and is basically just a tokenizer. These tokens are then fed through the expander, which expands any commands to more tokens. Finally, there is the execution step, in our case this step simply collects the final text in parts, potentially with any metadata containing information on how to process it in the HTML converter.

There are two main types of tokens: `CharacterToken` and `ControlSequenceToken`. Both of these are represented in our code by specializations of the `Token` class:

```
class Token
{
    public PositionInView position = new();

    public virtual void Expand(Expander exp) { }
    public virtual void Execute(Executor exe) { }
}
```

Each token remembers its position in the original text, this is useful when an error occurs, so we know exactly where to highlight the error in the text editor. A `CharacterToken` does not need to be expanded, and simply only defines the `Execute` method, which will append the character to the output. A `ControlSequenceToken` is a TEX or LATEX command, in its implementation of the `Expand` function, it first finds the appropriate command, and then calls `Expand` on that command. Its `Execute` implementation does a similar thing, but calls `Execute` instead.

The LaTeX parser is written in a very modular manner, every TeX and LaTeX command is its own class that extends the base class:

```
public class Command
{
    public virtual void Expand(Expander exp, Token self) { }
    public virtual void Execute(Executor exe, Token self) { }
}
```

These commands are either a TeX primitive, like `\par` or `\def`, or they are an implementation of a LaTeX command. Here is an example of the implementation of the LaTeX command `\title`:

```
class Title : Command
{
    public override void Execute(Executor exe, Token self)
    {
        var arg = exe.BalancedText(false, false, false);
        if (arg == null)
            throw new TeXception(self.position, "\\title expected argument");
        exe.GetScope().title = exe.TokensToString(arg);
    }
}
```

The title command does not need to `Expand`, it only defines an `Execute`. It first parses a `BalancedText`, this is defined in the TeX documentation as any sequence of tokens starting with an `opengroup` like '`{`', and ending in a `closegroup` like '`}`'. If this is parsed unsuccessfully, it will throw an exception, with the position of the token for error highlighting. If it was parsed successfully it will convert that `BalancedText` to a string, and assign it to the title.

When the parser has processed the entire input text, we are left with a list of `Parts`:

```
public class Part
{
    public enum Type { Paragraph, Math, Environment }
    public Type type = Type.Paragraph;
    public string value = "";
    public PartData? data = null;
}
```

These parts contain all the information to convert them into HTML, and text segments. Each part has one of three types, a `Paragraph` is normal text, optionally containing styling information in its `PartData`. The `Math` type contains an entire math block, in our case, this is the literal LaTeX, as we process the math LaTeX separately. Finally, the `Environment` type contains a `\begin` or `\end` command. The `value` field contains any text that is contained in this part. And finally, the `PartData` contains metadata, like the type of environment is opened or closed, what kind of section header this is etc. This fully depends on what command created this particular part.

6.2.2 Speech Generator

This project does not contain a text-to-speech engine so that leaves two options in order to get access to speech engines.

Browser Engine

First, the browser has a built-in text-to-speech engine, which has the advantage of being local and therefore requires no WiFi or other form of internet connection to work. This also means that the browser's engine will, on average, be faster. Since the data does not need to travel back and forth between the user and the server hosting the speech engine. Another advantage is that of privacy. By using the browser's built-in engine, files do not need to be sent to third-party software, which could save the information on the text people upload to our program. However, the speech engine in the browser has a low quality compared to an API and there is no reliable way to get the audio from the speech synthesizer. An implementation of the browser's speech synthesizer is quite simple and looks along the lines of the following:

```
// A function to turn the parameter 'text' into speech.
function textToSpeechBrowser(text) {
    const synth = window.speechSynthesis;
    const utter = new SpeechSynthesisUtterance(text);
    utter.rate = 0.8;
    synth.speak(utter);
}
```

Note that this example is one of the simplest methods of converting text to speech. The speech synthesis utterance has a number of variables that adjust the way the text is spoken. This includes the language of the utterance, the pitch and rate that the utterance will be spoken at, the voice used to speak the utterance, and the volume. (This link really needed?)(https://developer.mozilla.org/en-US/docs/Web/API/SpeechSynthesisUtterance).

API Connection

The second way of converting text to speech is using a third-party API. A third-party API brings both advantages and disadvantages. Most importantly, third-party software is very volatile, as they bring quality to the table, which is not something that can be said for the synthesizer embedded in the browser. Another benefit is that the existing software has multiple voices to choose from, which is pleasant for the user. On top of that, contrary to the browser's engine, APIs return an audio encoding, which can in turn be used to make a downloadable mp3 so the user can listen to the audio elsewhere or in another way, similar to an audiobook. There are a lot of choices already out there when it comes to speech engines, so the best one for the situation can be used. Some disadvantages include APIs requiring money for the computation of the audio. There is also a small decrease in privacy because the text within the file that the user uploads will have to be sent to the API in order to get the corresponding audio.

Audio can be requested from an API in the following manner:

```
// A function to turn the parameter 'text' into speech using
// the APIurl for the request destination.
async function textToSpeechAPI(text, APIurl) {
    let response = await fetch(APIurl, text, {*fetch options*});
    // Do something with the result depending on the API used...
}
```

Note that the function here is asynchronous because the code needs to wait until a response has been sent by the API.

Chapter 7

Testing the System

7.1 Test Plan

The test plan consists of several different parts and tests, starting with fixed testing moments. One fixed day each week in the planning is assigned to testing the system. This test day was jokingly called 'Testing Tuesday'. Testing Tuesday is planned as followed:

We created a planning, which had a list with 'ready for testing', containing all new features and bug fixes. All these features and fixes get tested and if they work correctly, they get moved to the 'done' list on the planning. If not, they get assigned to the 'bugs' list with a proper description of the problem and how it can be reproduced (if known). During the day, more time can be spent on fixing and testing these issues.

Furthermore, we plan to make use of Unit Testing for the LATEX parser, as explained in subsection 7.1.1 below. These tests will be written during Testing Tuesday as well. Last but not least, we will use Trial & Error Testing for both the LaTeX parser and the compiled preview and user interface. There will also be usability testing for testing and improving the general design. All testing methods are described below in more detail.

7.1.1 Unit Testing

Unit testing is a software testing method with the objective to isolate written code to test and determine if it works as intended [18]. The tested units are often the smallest testable and functional items or classes within the software. By testing these units (individually), "most of the errors that might be introduced into the code can be detected very early and prevented from propagating to other parts of the project." [19]

LaTeX Parser

Because of the decision to implement the LATEX parser ourselves, it is important to ensure it functions correctly, as it is the core of the project. Therefore unit tests will be written for each TEX and LATEX command that will be implemented. These unit tests

7.1.2 Trial & Error Testing

Because TeX and LaTeX have many different features, which causes many edge cases and combinations, it is impossible to cover everything with Unit Tests alone. Therefore the decision was made to develop several example files with as many different LaTeX features as we could think of. These files will be manually tested to find missing control sequences or other necessary parts.

Aside from creating new example files, older (anonymized) LATEX projects from group members will be used to see if both the parser and preview can handle the contents. Even when this method probably won't cover all possible features, it will at least help to find the most important ones.

7.1.3 Usability Testing

Our project group will have peer review meetings with other groups every 2 weeks. In this meeting, we will ask other groups to test the usability of our project and request feedback from their experiences. A group exists generally of 5 people, which is enough to get useful results according to Nielsen [20]. There will be 4 peer review meetings. The testing peers will probably not be provided with specific goals to achieve with our application, but the testers can use the application as they like and test all the features.

Furthermore, our team will also have a weekly meeting with the client by agreement. During these meetings, we will demonstrate the application to show the weekly progress. Based on these meetings we can receive feedback from the client about which features to add or change. The received feedback will be noted down to take into account while developing the system.

7.2 Test Results

The above-mentioned test plan yielded different results. These results are given below, including solutions for possible problems that were found while testing.

7.2.1 Implementation Test Results

LaTeX Parser

To ensure the parser works correctly, we implemented a set of unit tests covering each implemented LaTeX and TeX command. These unit tests were written by our team in a separate utility class. The tests cover the basic usage and functionality of the supported commands. Not all edge cases are handled by these tests because of the broad diversity in LaTeX commands, but the most important edge cases are covered by our trial and error testing. All unit tests passed, indicating that the parser correctly handles the basic usage of each supported command.

An example of the results and output of the unit tests is given in appendix B.1.

7.2.2 User Experience Test Results

The testers from the peer review sessions did not give much useful feedback over the four sessions. Our requirements were good according to the first peer review group.

The second group was wondering whether making your own parser would be worth the time, or whether our project group should have rather used an already existing parser, saving us time. On the first prototypes, we provided them they had little commentary, except that it 'looked good' and that they appreciated the thought and work in making the interface more accessible and disability friendly. After the parser commentary, we had another group discussion on the parser but made the decision to stay on the same course. This was because our parser was already functional and working at that point in time, and we still saw the advantages of having more control over the parser. We did however agree to put more focus on the TTS part of the system.

The third group had no feedback at all, and the fourth group merely peer-reviewed the posters. Concluding: We should have been more specific with the tests we gave them and asked them for more specific feedback, such that we would have gotten more useful replies.

Our client did give a lot of useful feedback each week. Some examples: The client said that we should consider whether we want to support math, code, or both. And when both were implemented, she advised us to look at a better transition between math, code, and text. Next, she told us to think about using a natural-sounding voice, instead of a more robotic voice from an earlier version. She also requested a document with info on which hardware specs are needed to run the application. Last but not least, when showing the client the disability-friendly styling, she advised even more color customizability.

All of the mentioned feedback above has been taken into account. A more natural-sounding voice was implemented, and the design/interface has many color and styling options. Syntax highlighting for Python code is supported as well based on the feedback, and better pronunciation of the parsed math was implemented. Some more added features suggested by the client, later on, were: a download audio button, support in many languages, and the ability to change the rate of speech.

7.2.3 Performance Testing

While originally no formal performance testing was conducted on the system, at one point a performance issue was identified. Specifically, at a certain point, the text-to-speech API we were using was taking a long time to load. So we decided to try a different approach, to minimize waiting time. The solution was to divide the text into smaller segments and load those asynchronously. When a segment is being loaded it will be blurred in the preview, to give some feedback to the user and let them know it is currently loading. If a user were to click on a segment that has not yet been scheduled to load, it will prioritize that segment. This approach minimizes the waiting time for the user.

Besides this, the system seems to perform fine, and with reasonable waiting times for the user, there aren't any major bottlenecks and it generally feels responsive. With slightly larger files the parser will of course take slightly longer, but this is not a significant bottleneck.

Chapter 8

Future Planning

8.1 Utilization & Support

8.1.1 documentation

LaTeX Parser

As mentioned in section 6.2.1, the parser is written in a very modular manner. Most implemented TeX primitives contain documentation comments that include a link to TeX by Topic [21], this book contains a comprehensive explanation of the functionality of the respective TeX primitive. The different stages of the processor, as mentioned in section 6.2.1, contain documentation comments explaining what the step does, also with a link to the respective chapter in TeX by Topic [21]. Sprinkled around the code are also `TODO` comments explaining in detail what edge cases are not handled yet, again with a link to the respective chapter in TeX by Topic [21].

In general, every aspect of the parser is explained with comments, often with a reference to TeX by Topic [21].

8.1.2 Front-end

Just as with the LaTeX parser, the code in the front-end part of the project has comments above most of the functions explaining what they do. Whenever necessary, the code also contains extra comments within the functions for more clarity. The files are neatly organised and named to match their corresponding page.

8.2 Future Expansion Possibilities

8.2.1 Collaborative editing

One of the features Overleaf has over our own editor is that multiple people can edit the document at the same time. This feature would also be useful for people that would need a text-to-speech function like the one our application offers.

8.2.2 Customisable hotkeys

While our current application is, styling-wise, about as customizable as you can get, (bar editing the CSS files manually,) we don't have an option for customizing the keyboard shortcuts that we offer with our application. Customising keyboard shortcuts can help for people who want to avoid using the mouse but find the current shortcuts hard to reach.

8.2.3 Multiple files

Large latex projects can get unwieldy if they are contained in a single file. To help organize large documents, LaTeX normally uses the `\include` command to include file contents in a bigger file when compiling.

8.2.4 Custom Text to Speech engine

As discussed in 6.1.1, we have found some shortcomings in the default browser TTS engine. However, since we found speechify [5] not very responsible to use we decided to go with that implementation for now anyways, with an API to quickly swap out the browser TTS engine if necessary (see 6.2.2). Of course, if there was more time, there could also be the option of making that TTS engine ourselves.

Chapter 9

Discussion & Conclusion

9.1 Evaluation

9.1.1 Planning

At the start of the project, we decided to follow a weekly schedule:

Monday	Tuesday	Wednesday	Thursday	Friday
Physical Stand-up	Online Stand-up	Physical Stand-up	Online Stand-up	Online Stand-up
Sprint Planning	Testing Tuesday	Meeting with supervisor		Sprint Review
Every 2 weeks peer review				Finalizing Friday

We mostly kept to this schedule throughout the project, with a couple of minor deviations. We did not end up meeting with the supervisor every week, and some weeks testing Tuesday was shuffled around a bit and did not end up on a Tuesday. To keep ourselves organized we decided to have a finalizing Friday, where we summarize the week and write down some notes. We kept that up for most of the weeks.

9.1.2 Team Responsibilities

Besides the initial design aspect, the project was divided into these parts: Infrastructure, back-end, parser, (User) Interface and (Lo-Fi) prototype design; Infrastructure, back-end, parser, (User) Interface and TTS development; Requirement analysis, background/problem research and requirement specification; and testing and test designs. Of course, there were also responsibilities such as communication with the client/supervisor and writing the deliverables. This was the general task division between team members:

Team Member	Tasks
Beitske Flake	Requirement Analysis & Background Research, UML Diagrams Creator, Interface Designer, (Lo-Fi) Prototype Designer, assisting in Interface Development, fine-tuning math to speech, Tester
Erik Oosting	(Lo-Fi) Prototype Designer, assisting in Interface Development, assisting in Parser / Back-end development
Jeroen Zwiers	Infrastructure Designer & Developer, Parser / Back-end Developer, assisting in Interface Development, assisting in TTS Development
Joris Bosman	Infrastructure Designer, Interface Designer & Developer, TTS Developer, assisting in Parser / Back-end development
Xander Heij	UML Diagrams Creator, Interface Designer, Tester, Test Designer, assisting in Interface Development, fine-tuning math to speech

There were some general responsibilities and tasks that were a group effort. This includes communication with the client/supervisor as well as contributing to project deliverables such as the reports and the manual. Moreover, everyone contributed to the creation of the poster and the slides for the presentations (both the final presentation as well as the peer reviews). Large parts of the infrastructure designing, requirement specification and planning were a group decision and effort as well.

9.1.3 Team Evaluation

The teamwork during this project went smoothly. As mentioned in section 9.1.1, we did daily standups and kept those up pretty much entirely throughout the project. Communication was not an issue, we made weekly plans, and everyone knew what to work on, and who to go to for specific questions. Task delegation was therefore also not a problem, our well-kept Trello board made it easy to divide the tasks and requirements among team members.

9.2 Conclusion

In the end, we managed to deliver a solid web application that converts a LaTeX document into speech. It has many features such as a preview with clickable and highlighted segments, a LaTeX text editor, a highly customizable user interface, a choice of many different voices, and many more. Our objectives, as mentioned in chapter 1.2, were definitely achieved. The text of the compiled LaTeX is spoken aloud to the user, including mathematical formulas, and (pseudo) code. And our user interface contains many features to be highly accessible to people with a visual disability. And finally, of course, we had loads of fun designing, building, and testing the LaTeXt to Speech application.

Bibliography

[1] Alberto Pepe. How many scholarly articles are written in latex? *Authorea Preprints*, 2017.

[2] Sheryl M Handler, Walter M Fierson, Section on Ophthalmology, American Association for Pediatric Ophthalmology Council on Children with Disabilities, American Academy of Ophthalmology, Strabismus, and American Association of Certified Orthoptists. Learning disabilities, dyslexia, and vision. *Pediatrics*, 127(3):e818–e856, 2011.

[3] Poulomee Datta and Joy Talukdar. The impact of vision impairment on students' self-concept. *International Journal of Inclusive Education*, 20(6):659–672, 2016.

[4] Piotr Brzoza and Dominik Spinczyk. Multimedia browser for internet online daisy books. In *Computers Helping People with Special Needs: 10th International Conference, ICCHP 2006, Linz, Austria, July 11-13, 2006. Proceedings 10*, pages 1087–1093. Springer, 2006.

[5] Cliff Weitzman. Speechify: Best free text to speech voice reader, Mar 2023. URL https://speechify.com/.

[6] Volker Sorge. Convert LaTeX or MathML to Speech, 5 2022. URL https://mathjax.github.io/MathJax-demos-web/speech-generator/convert-with-speech.html.

[7] Taichen Rose, Walker Herring, and Connor Barlow. Tex2Speech, 7 2021. URL https://tex2speech-website.vercel.app/.

[8] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. Manifesto for agile software development. 2001.

[9] Ken Schwaber. Home, 6 2010. URL https://www.scrum.org/.

[10] Atlassian. Manage Your Team's Projects From Anywhere — Trello, 2017. URL https://trello.com/.

[11] Kevin Brennan et al. *"MoSCoW Analysis" - A Guide to the Business Analysis Body of Knowledger*, volume 6.1.5.2. Iiba, 2 edition, 2009. ISBN 978-0-9811292-1-1.

[12] Mark J Price. *C# 9 and. NET 5–Modern Cross-Platform Development: Build intelligent apps, websites, and services with Blazor, ASP. NET Core, and Entity Framework Core using Visual Studio Code*. Packt Publishing Ltd, 2020.

[13] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[14] British Dyslexia Association. Dyslexia friendly style guide - British Dyslexia Association, 2023. URL https://www.bdadyslexia.org.uk/advice/employers/creating-a-dyslexia-friendly-workplace/dyslexia-friendly-style-guide.

[15] Luz Rello and Jeffrey P Bigham. Good background colors for readers: A study of people with and without dyslexia. In *Proceedings of the 19th international ACM SIGACCESS conference on computers and accessibility*, pages 72–80, 2017.

[16] Klaus Pohl. *Requirements engineering fundamentals: a study guide for the certified professional for requirements engineering exam-foundation level-IREB compliant*. Rocky Nook, Inc., 2016.

[17] Arno Gourdol. Arnog/mathlive: A web component for easy math input, Apr 2021. URL https://github.com/arnog/mathlive/.

[18] TechTarget Contributor. unit testing. *Software Quality*, 8 2019. URL https://www.techtarget.com/searchsoftwarequality/definition/unit-testing.

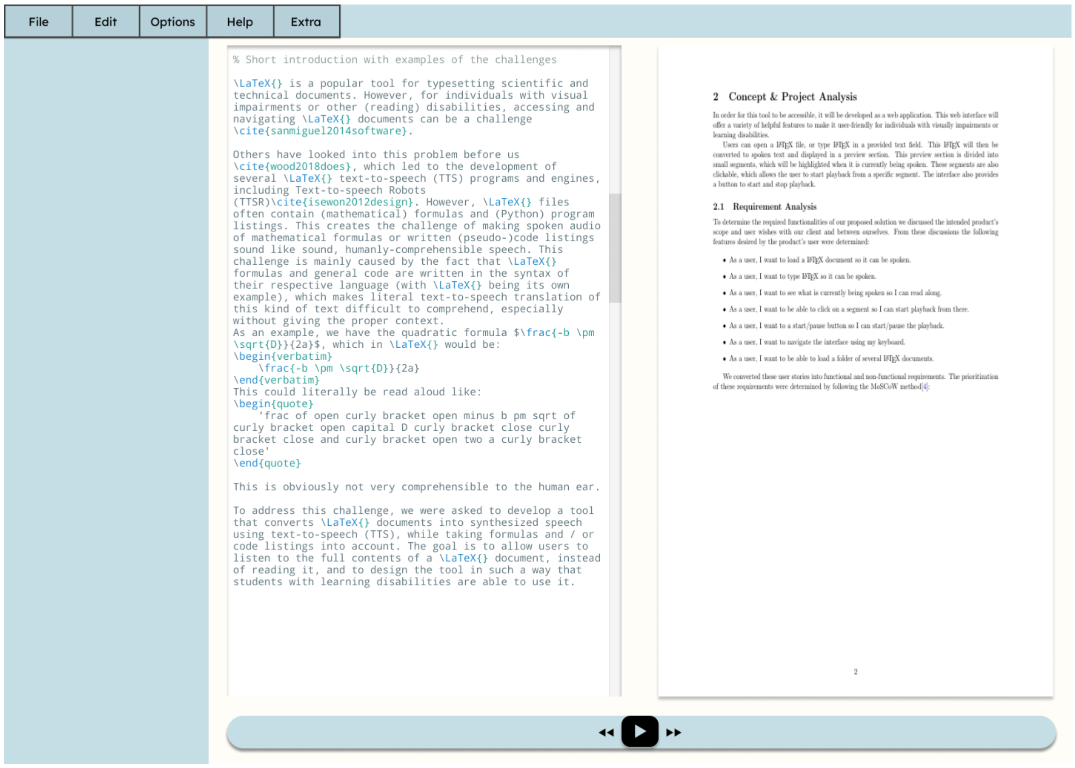
[19] Dorota Huizinga and Adam Kolawa. *Automated defect prevention: best practices in software management*. John Wiley & Sons, 2007.

[20] Jakob Nielsen and Thomas K Landauer. A mathematical model of the finding of usability problems. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 206–213, 1993.

[21] Victor Eijkhout. *TeX by Topic: A TeXnician's Reference*. 2 1992.

Appendix A

Mock-ups



```
% Short introduction with examples of the challenges

\LaTeX{} is a popular tool for typesetting scientific and
technical documents. However, for individuals with visual
impairments or other (reading) disabilities, accessing and
navigating \LaTeX{} documents can be a challenge
\cite{sanmiguel2014software}.

Others have looked into this problem before us
\cite{wood2018does}, which led to the development of
several \LaTeX{} text-to-speech (TTS) programs and engines,
including Text-to-speech Robots
(TTSR)\cite{isewon2012design}. However, \LaTeX{} files
often contain (mathematical) formulas and (Python) program
listings. This creates the challenge of making spoken audio
of mathematical formulas or written (pseudo-)code listings
sound like sound, humanly-comprehensible speech. This
challenge is mainly caused by the fact that \LaTeX{}
formulas and general code are written in the syntax of
their respective language (with \LaTeX{} being its own
example), which makes literal text-to-speech translation of
this kind of text difficult to comprehend, especially
without giving the proper context.
As an example, we have the quadratic formula $\frac{-b \pm
\sqrt{D}}{2a}$, which in \LaTeX{} would be:
\begin{verbatim}
    \frac{-b \pm \sqrt{D}}{2a}
\end{verbatim}
This could literally be read aloud like:
\begin{quote}
    'frac of open curly bracket open minus b pm sqrt of
curly bracket open capital D curly bracket close curly
bracket close and curly bracket open two a curly bracket
close'
\end{quote}

This is obviously not very comprehensible to the human ear.

To address this challenge, we were asked to develop a tool
that converts \LaTeX{} documents into synthesized speech
using text-to-speech (TTS), while taking formulas and / or
code listings into account. The goal is to allow users to
listen to the full contents of a \LaTeX{} document, instead
of reading it, and to design the tool in such a way that
students with learning disabilities are able to use it.
```

2 Concept & Project Analysis

In order for this tool to be accessible, it will be developed as a web application. This web interface will offer a variety of helpful features to make it user-friendly for individuals with visually impairments or learning disabilities.

Users can open a LaTeX file, or type LaTeX in a provided text field. This LaTeX will then be converted to spoken text and displayed in a preview section. This preview section is divided into small segments, which will be highlighted when it is currently being spoken. These segments are also clickable, which allows the user to start playback from a specific segment. The interface also provides a button to start and stop playback.

2.1 Requirement Analysis

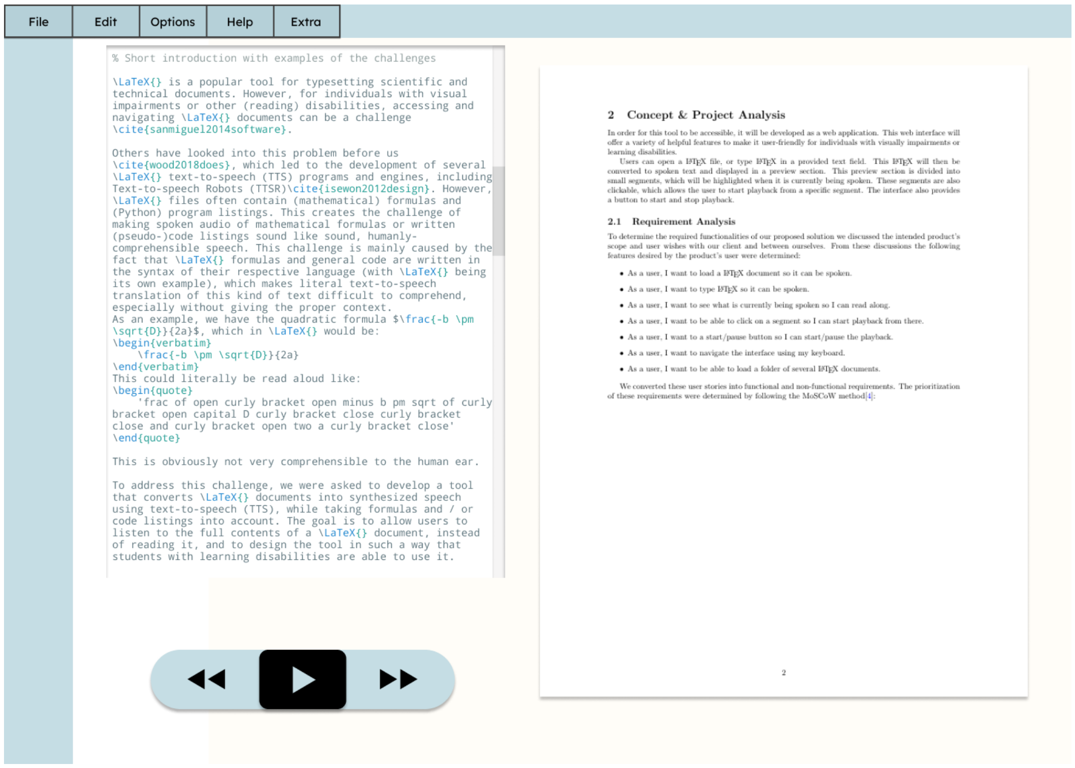
To determine the required functionalities of our proposed solution we discussed the intended product's scope and user wishes with our client and between ourselves. From these discussions the following features desired by the product's user were determined:

- As a user, I want to load a LaTeX document so it can be spoken.
- As a user, I want to type LaTeX so it can be spoken.
- As a user, I want to see what is currently being spoken so I can read along.
- As a user, I want to be able to click on a segment so I can start playback from there.
- As a user, I want to a start/pause button so I can start/pause the playback.
- As a user, I want to navigate the interface using my keyboard.
- As a user, I want to be able to load a folder of several LaTeX documents.

We converted these user stories into functional and non-functional requirements. The prioritization of these requirements were determined by following the MoSCoW method[4]:

2

Figure A.1: First prototype A



```
% Short introduction with examples of the challenges

\LaTeX{} is a popular tool for typesetting scientific and
technical documents. However, for individuals with visual
impairments or other (reading) disabilities, accessing and
navigating \LaTeX{} documents can be a challenge
\cite{sanmiguel2014software}.

Others have looked into this problem before us
\cite{wood2018does}, which led to the development of several
\LaTeX{} text-to-speech (TTS) programs and engines, including
Text-to-speech Robots (TTSR)\cite{isewon2012design}. However,
\LaTeX{} files often contain (mathematical) formulas and
(Python) program listings. This creates the challenge of
making spoken audio of mathematical formulas or written
(pseudo-)code listings sound like sound, humanly-
comprehensible speech. This challenge is mainly caused by the
fact that \LaTeX{} formulas and general code are written in
the syntax of their respective language (with \LaTeX{} being
its own example), which makes literal text-to-speech
translation of this kind of text difficult to comprehend,
especially without giving the proper context.
As an example, we have the quadratic formula $\frac{-b \pm
\sqrt{D}}{2a}$, which in \LaTeX{} would be:
\begin{verbatim}
    \frac{-b \pm \sqrt{D}}{2a}
\end{verbatim}
This could literally be read aloud like:
\begin{quote}
    'frac of open curly bracket open minus b pm sqrt of curly
bracket open capital D curly bracket close curly bracket
close and curly bracket open two a curly bracket close'
\end{quote}

This is obviously not very comprehensible to the human ear.

To address this challenge, we were asked to develop a tool
that converts \LaTeX{} documents into synthesized speech
using text-to-speech (TTS), while taking formulas and / or
code listings into account. The goal is to allow users to
listen to the full contents of a \LaTeX{} document, instead
of reading it, and to design the tool in such a way that
students with learning disabilities are able to use it.
```

2 Concept & Project Analysis

In order for this tool to be accessible, it will be developed as a web application. This web interface will offer a variety of helpful features to make it user-friendly for individuals with visually impairments or learning disabilities.

Users can open a LaTeX file, or type LaTeX in a provided text field. This LaTeX will then be converted to spoken text and displayed in a preview section. This preview section is divided into small segments, which will be highlighted when it is currently being spoken. These segments are also clickable, which allows the user to start playback from a specific segment. The interface also provides a button to start and stop playback.

2.1 Requirement Analysis

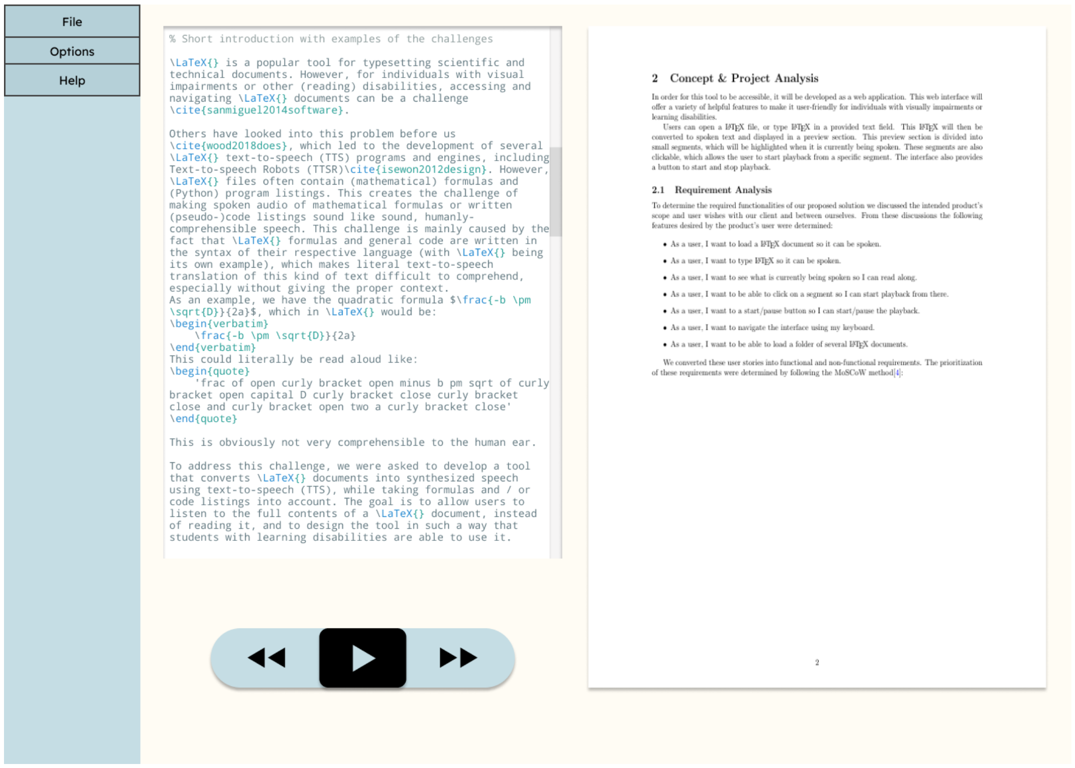
To determine the required functionalities of our proposed solution we discussed the intended product's scope and user wishes with our client and between ourselves. From these discussions the following features desired by the product's user were determined:

- As a user, I want to load a LaTeX document so it can be spoken.
- As a user, I want to type LaTeX so it can be spoken.
- As a user, I want to see what is currently being spoken so I can read along.
- As a user, I want to be able to click on a segment so I can start playback from there.
- As a user, I want to a start/pause button so I can start/pause the playback.
- As a user, I want to navigate the interface using my keyboard.
- As a user, I want to be able to load a folder of several LaTeX documents.

We converted these user stories into functional and non-functional requirements. The prioritization of these requirements were determined by following the MoSCoW method[4]:

2

Figure A.2: First prototype B



```
% Short introduction with examples of the challenges

\LaTeX{} is a popular tool for typesetting scientific and
technical documents. However, for individuals with visual
impairments or other (reading) disabilities, accessing and
navigating \LaTeX{} documents can be a challenge
\cite{sanmiguel2014software}.

Others have looked into this problem before us
\cite{wood2018does}, which led to the development of several
\LaTeX{} text-to-speech (TTS) programs and engines, including
Text-to-speech Robots (TTSR)\cite{isewon2012design}. However,
\LaTeX{} files often contain (mathematical) formulas and
(Python) program listings. This creates the challenge of
making spoken audio of mathematical formulas or written
(pseudo-)code listings sound like sound, humanly-
comprehensible speech. This challenge is mainly caused by the
fact that \LaTeX{} formulas and general code are written in
the syntax of their respective language (with \LaTeX{} being
its own example), which makes literal text-to-speech
translation of this kind of text difficult to comprehend,
especially without giving the proper context.
As an example, we have the quadratic formula $\frac{-b \pm
\sqrt{D}}{2a}$, which in \LaTeX{} would be:
\begin{verbatim}
    \frac{-b \pm \sqrt{D}}{2a}
\end{verbatim}
This could literally be read aloud like:
\begin{quote}
    'frac of open curly bracket open minus b pm sqrt of curly
bracket open capital D curly bracket close curly bracket
close and curly bracket open two a curly bracket close'
\end{quote}

This is obviously not very comprehensible to the human ear.

To address this challenge, we were asked to develop a tool
that converts \LaTeX{} documents into synthesized speech
using text-to-speech (TTS), while taking formulas and / or
code listings into account. The goal is to allow users to
listen to the full contents of a \LaTeX{} document, instead
of reading it, and to design the tool in such a way that
students with learning disabilities are able to use it.
```

2 Concept & Project Analysis

In order for this tool to be accessible, it will be developed as a web application. This web interface will offer a variety of helpful features to make it user-friendly for individuals with visually impairments or learning disabilities.

Users can open a LaTeX file, or type LaTeX in a provided text field. This LaTeX will then be converted to spoken text and displayed in a preview section. This preview section is divided into small segments, which will be highlighted when it is currently being spoken. These segments are also clickable, which allows the user to start playback from a specific segment. The interface also provides a button to start and stop playback.

2.1 Requirement Analysis

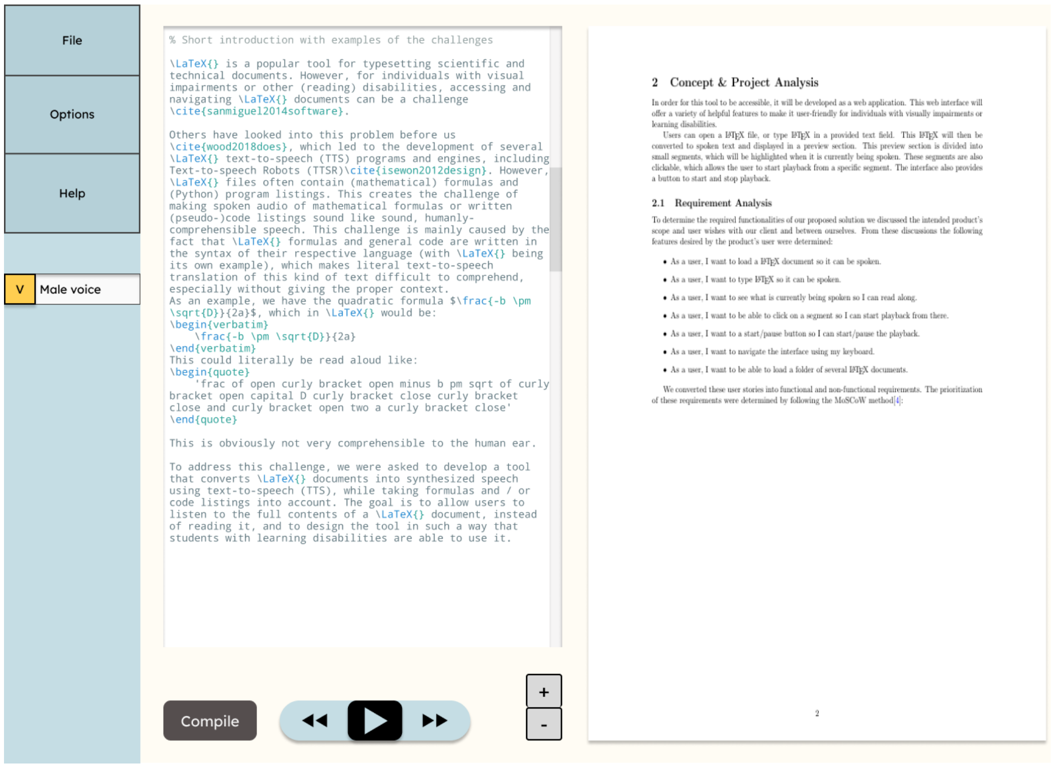
To determine the required functionalities of our proposed solution we discussed the intended product's scope and user wishes with our client and between ourselves. From these discussions the following features desired by the product's user were determined:

- As a user, I want to load a LaTeX document so it can be spoken.
- As a user, I want to type LaTeX so it can be spoken.
- As a user, I want to see what is currently being spoken so I can read along.
- As a user, I want to be able to click on a segment so I can start playback from there.
- As a user, I want to a start/pause button so I can start/pause the playback.
- As a user, I want to navigate the interface using my keyboard.
- As a user, I want to be able to load a folder of several LaTeX documents.

We converted these user stories into functional and non-functional requirements. The prioritization of these requirements were determined by following the MoSCoW method[4]:

2

Figure A.3: Improved prototype A



```
% Short introduction with examples of the challenges

\LaTeX{} is a popular tool for typesetting scientific and
technical documents. However, for individuals with visual
impairments or other (reading) disabilities, accessing and
navigating \LaTeX{} documents can be a challenge
\cite{sanmiguel2014software}.

Others have looked into this problem before us
\cite{wood2018does}, which led to the development of several
\LaTeX{} text-to-speech (TTS) programs and engines, including
Text-to-speech Robots (TTSR)\cite{isewon2012design}. However,
\LaTeX{} files often contain (mathematical) formulas and
(Python) program listings. This creates the challenge of
making spoken audio of mathematical formulas or written
(pseudo-)code listings sound like sound, humanly-
comprehensible speech. This challenge is mainly caused by the
fact that \LaTeX{} formulas and general code are written in
the syntax of their respective language (with \LaTeX{} being
its own example), which makes literal text-to-speech
translation of this kind of text difficult to comprehend,
especially without giving the proper context.
As an example, we have the quadratic formula $\frac{-b \pm
\sqrt{D}}{2a}$, which in \LaTeX{} would be:
\begin{verbatim}
    \frac{-b \pm \sqrt{D}}{2a}
\end{verbatim}
This could literally be read aloud like:
\begin{quote}
    'frac of open curly bracket open minus b pm sqrt of curly
bracket open capital D curly bracket close curly bracket
close and curly bracket open two a curly bracket close'
\end{quote}

This is obviously not very comprehensible to the human ear.

To address this challenge, we were asked to develop a tool
that converts \LaTeX{} documents into synthesized speech
using text-to-speech (TTS), while taking formulas and / or
code listings into account. The goal is to allow users to
listen to the full contents of a \LaTeX{} document, instead
of reading it, and to design the tool in such a way that
students with learning disabilities are able to use it.
```

2 Concept & Project Analysis

In order for this tool to be accessible, it will be developed as a web application. This web interface will offer a variety of helpful features to make it user-friendly for individuals with visually impairments or learning disabilities.

Users can open a LaTeX file, or type LaTeX in a provided text field. This LaTeX will then be converted to spoken text and displayed in a preview section. This preview section is divided into small segments, which will be highlighted when it is currently being spoken. These segments are also clickable, which allows the user to start playback from a specific segment. The interface also provides a button to start and stop playback.

2.1 Requirement Analysis

To determine the required functionalities of our proposed solution we discussed the intended product's scope and user wishes with our client and between ourselves. From these discussions the following features desired by the product's user were determined:

- As a user, I want to load a LaTeX document so it can be spoken.
- As a user, I want to type LaTeX so it can be spoken.
- As a user, I want to see what is currently being spoken so I can read along.
- As a user, I want to be able to click on a segment so I can start playback from there.
- As a user, I want to a start/pause button so I can start/pause the playback.
- As a user, I want to navigate the interface using my keyboard.
- As a user, I want to be able to load a folder of several LaTeX documents.

We converted these user stories into functional and non-functional requirements. The prioritization of these requirements were determined by following the MoSCoW method[4]:

2

Figure A.4: Extra features for prototype A

Appendix B

Test Results

B.1 Unit Testing

An example of a test for the TeX primitive `\def`, using our custom `TestUtils.DoTest` function:

```
TestUtils.DoTest(
    /* Name of test    */ @"\def, 2 Arguments With Delimiters",
    /* LaTeX to test   */ @"\def\test(#1,#2){#1#2} \test(Hello, World)",
    /* Expected output */ "Hello World",
    /* Verbose print   */ true
);
```

The verbose console output:

```
=== Beginning test [\def, 2 Arguments With Delimiters] ===
$ Parsed: 1:0: Control Sequence [def]
+ Expanding: 1:0: Control Sequence [def]
* Executing: 1:0: Control Sequence [def]
  $ Parsed: 1:4: Control Sequence [test]
  $ Parsed: 1:9: Character [(]
  $ Parsed: 1:10: Character [#]
  $ Parsed: 1:11: Character [1]
  $ Parsed: 1:12: Character [,]
  $ Parsed: 1:13: Character [#]
  $ Parsed: 1:14: Character [2]
  $ Parsed: 1:15: Character [)]
  $ Parsed: 1:16: Character [{]
  $ Parsed: 1:17: Character [#]
  $ Parsed: 1:18: Character [1]
  $ Parsed: 1:19: Character [#]
  $ Parsed: 1:20: Character [2]
  $ Parsed: 1:21: Character [}]
$ Parsed: 1:22: Character [ ]
+ Expanding: 1:22: Character [ ]
* Executing: 1:22: Character [ ]
$ Parsed: 1:23: Control Sequence [test]
+ Expanding: 1:23: Control Sequence [test]
  $ Parsed: 1:28: Character [(]
  $ Parsed: 1:29: Character [H]
  $ Parsed: 1:30: Character [e]
  $ Parsed: 1:31: Character [l]
  $ Parsed: 1:32: Character [l]
  $ Parsed: 1:33: Character [o]
  $ Parsed: 1:34: Character [,]
  $ Parsed: 1:35: Character [ ]
  $ Parsed: 1:36: Character [W]
  $ Parsed: 1:37: Character [o]
  $ Parsed: 1:38: Character [r]
  $ Parsed: 1:39: Character [l]
  $ Parsed: 1:40: Character [d]
  $ Parsed: 1:41: Character [)]
+ Expanding: 1:29: Character [H]
* Executing: 1:29: Character [H]
+ Expanding: 1:30: Character [e]
* Executing: 1:30: Character [e]
+ Expanding: 1:31: Character [l]
* Executing: 1:31: Character [l]
+ Expanding: 1:32: Character [l]
* Executing: 1:32: Character [l]
+ Expanding: 1:33: Character [o]
* Executing: 1:33: Character [o]
+ Expanding: 1:35: Character [ ]
* Executing: 1:35: Character [ ]
+ Expanding: 1:36: Character [W]
* Executing: 1:36: Character [W]
+ Expanding: 1:37: Character [o]
* Executing: 1:37: Character [o]
+ Expanding: 1:38: Character [r]
* Executing: 1:38: Character [r]
+ Expanding: 1:39: Character [l]
* Executing: 1:39: Character [l]
+ Expanding: 1:40: Character [d]
* Executing: 1:40: Character [d]
$ Parsed: 1:42: Character [ ]
+ Expanding: 1:42: Character [ ]
* Executing: 1:42: Character [ ]
$ Parsed:
Passed test [\def, 2 Arguments With Delimiters]
```