Vasista Vovveti

Project: Presentation Generator

Project Report

Overview:

Open-source projects often host their own documentation online. Speakers and lecturers use this documentation as a source to create presentations as guides for their talks and lessons. While, this workflow initially works, it falls apart over time. The presentations go out of date and adequate effort is not made to update the slides.

An example of this is the documentation for the FIRST Robotics Competition (FRC). FRC is a high school level robotics competition. High school students in this competition build, wire, and program robots. To cover the hardware and software components FRC uses, it has a website containing extensive documentation. Coaches of local teams take information from this documentation to create presentations for lessons to teach their students. But, every year, the hardware available and the software APIs slightly change. Coaches don't always have the manpower to keep up and update all of their presentations, especially as their life and careers take time away from volunteering.

My solution to this problem is to convert documentation webpages into presentations. Presentations will be generated client side. This way, presentations can never be out of date. They will always be in line with the contents of the webpage. Furthermore, this doesn't add any extra complexity for open-source project maintainers.

Implementation:

My code has 4 major steps: Generate ADT, Split Paragraphs, Render Slides, and Re-render Slides.

The Generate ADT step takes the HTML elements of the source webpage and creates an ADT out of it. My top-level types of my ADT are Presentation, Slide, and Item. A Presentation contains Slides. Slides contain Items. Items are a union of types Html, Figure, Text, and Slide. Since Items can be Slides, this ADT is recursive. Depending on the type of html read, different Items are populated. Paragraph elements populate Text Items. Image elements populate Figure Items. All other elements are passed directly as HTML via Html Items. This step was listed in the project proposal and is implemented as written.

The split paragraphs step takes paragraphs and splits them into individual sentences. This is needed for readability. Documentation is written in paragraphs while presentations are written in bullet points. Paragraphs get split into sentences and each sentence becomes a bullet point on the slide. I split paragraphs using a lexer and parser. I created rules for the lexer that generate tokens Space, Parenthesis, Quote, Punctuation, EOF, and Word. These tokens are then parsed to create sentences. Each time a Punctuation or EOF token is parsed, a new sentence begins. This step was not listed in the project proposal. When starting on the implementation for this project, I had not envisioned that large blocks of text would be an issue. I added this step in later to make the presentations more readable.

The render slides step takes the ADT for a Slide and renders it into HTML. As Slides consist of Items, Items are first rendered into HTML. Then, the Slide renders while laying out the Items it contains. I implemented 3 Slide types, “title”, “title and content”, and “figure and content”. Each of these slide types has a different layout. I pattern match on each type of Slide and Item to have custom HTML for each one. This step was listed in the project proposal. I have implemented the same Slide and Item types as listed in the proposal. Though, I have added in one extra type of Item, Html. Html is treated as the default case for when a webpage HTML element cannot be converted into a Text or Figure.

The re-render step takes slides and splits and re-renders them if necessary. If the viewer’s web browser window is too small to show the contents of a slide, then it is split into 2 slides so that all content is visible on screen. This step was in the project proposal as “dynamic slides”.

All the components listed above work well. They work on live websites and generate full presentations. The only part of the project proposal not implemented are static slides. Static slides were proposed as slides with fixed dimensions as opposed to dynamic slides, which automatically re-layout. I did not implement static slides as it’s implementation would take too much custom CSS and HTML work. I would have had to fix font sizes, and the browser zoom level. I chose to spend time on the new split paragraphs step instead as that more meaningfully improved the quality of the presentations.

Implementation – Code Layout:

My code is implemented in 4 files, element.ts, utils.ts, split.ts, and present.ts. element.ts contains an abstraction for an HTML element that adds extra methods and syntactic sugar. This makes the code in other files more readable. utils.ts contains helper functions for creating HTML and Javascript elements. This makes injecting code into the source webpage easier. split.ts contains the code used for splitting paragraphs into sentences. It contains the rules for the lexer and contains the parser. present.ts contains the code to build up the ADT, render the ADT, and initialize the presentation.

Tests:

My code includes 71 test cases, including unit tests, feature tests, and an end-to-end test. All the utility functions in utils.ts are unit tested to make sure that the functions return the correct values based on various inputs. As these functions are used throughout the rest of the codebase, testing these is critical in making sure bugs don’t propagate. I create HTML and Javascript elements with various inputs and make sure the elements contain the correct attributes.

All the methods in element.ts are similarly tests for the same reasons as above. My abstraction for HTML elements is used extensively in ADT generation and rendering.

The functions in split.ts are unit tested and feature tested. The lexer is unit tested to make sure that it produces the correct tokens for various inputs. The parser is unit tested to make sure that it produces the correct phrases given. Finally, the whole lexer + parser combination is feature tested to make sure that the correct sentences are generated given an entire paragraph. The paragraphs passed in to these feature tests come directly from documentation, namely the OCaml documentation and FRC documentation. These tests cover the split paragraphs phase.

The render and re-render steps are tested. Each substep (render Item, render Slide, and split Slide) is tested separately across various sizes of Slide, various types of Slide, and various types of Item.

I also have an end-to-end test to ensure that the presentation generates correctly and is usable. The test opens a web browser, navigates to a documentation site, injects code, starts the presentation, and navigates through the slides. This makes sure the project as a whole is in working condition.

Code Listing:

Code listing is the rest of this report.
Note on running this codebase:
As this is a typescript project, it depends on npm. A makefile is not sufficient to run everything.
- `npm install` installs all the dependencies for this project
- `npm test` runs all tests
- `npx webpack` compiles the code targeting web browser Javascript

The end-to-end test opens up a full Chrome browser. For this test case, the Chrome Webdriver must be downloaded from Google's website.

present.ts

```typescript
"use strict";
export { present, buildPresentation, buildPresentationCPS, splitSlide, splitSlideCPS,
buildItem, buildItemCPS, buildSlide, buildSlideCPS, Slide, Item, Text, Html, Figure,
Presentation };

import Reveal from 'reveal.js';
import { match, __, not, select, when } from 'ts-pattern';

import {H, Hwrappable, _H} from './element';
import {splitParagraph} from './split';
import { createScriptElement, createScriptSrcElement, getNextSiblings, htmlToElement} from
'./utils';

// ----- ADT Types -----

type Html =
    | { type: "html", h: _H }

type Figure =
    | { type: "figure", h: _H }

type Text =
    | { type: "text", text: string}

type Item = Html | Figure | Text | Slide

type Slide =
    { items: Item[], path: string[] }
    & (
        | { type: "title" }
        | { type: "title and content" }
        | { type: "figure and content" }
    )

type Presentation =
    | { slides: Slide[] }


// Presentation container
let presentation: Presentation = { slides: [] };
(window as any).presentation = presentation;

function splitSlide(slide: Slide): Slide[] {
    /* Splits 1 slide into 2 slides if possible. If not, returns the input. */
    return match(slide)
        .with({type: "title"}, () => [slide])
        .with({type: "title and content"}, () => {
            // small slides shouldn't be split
            if (slide.items.length <= 2) {
                return [slide];
            }

            // Deep copy slide
            let slide1 = JSON.parse(JSON.stringify(slide));
            let slide2 = JSON.parse(JSON.stringify(slide));

            // Split all but first item (title)
            slide1.items = slide.items.slice(1, (slide.items.length - 1) / 2 + 1);
            slide2.items = slide.items.slice((slide.items.length - 1) / 2 + 1);

            return match(slide.items[0])
                .with({type: "text"}, (item) => {
                    slide1.items.unshift({type: "text", text: item.text});
                    slide2.items.unshift({type: "text", text: item.text});
                    return [slide1, slide2];
                })
                .otherwise(() => [slide]);
        })
        .with({type: "figure and content"}, () => {
            // small slides shouldn't be split
            if (slide.items.length == 2) {
                return [slide];
            }

            // Deep copy slide
            let slide1 = JSON.parse(JSON.stringify(slide));
            let slide2 = JSON.parse(JSON.stringify(slide));

            // Split all but first item (figure)
            slide1.items = slide.items.slice(1, (slide.items.length - 1) / 2 + 1);
            slide2.items = slide.items.slice((slide.items.length - 1) / 2 + 1);

            return match(slide.items[0])
                .with({type: "figure"}, (item) => {
                    // JSON doesn't copy HTML elements, so we need to copy them manually
                    slide1.items.unshift({type: "figure", h: H(item.h.deepcopy())});
                    slide2.items.unshift({type: "figure", h: H(item.h.deepcopy())});
                    return [slide1, slide2];
                })
                .otherwise(() => [slide]);
        })
        .exhaustive();
}
(window as any).splitSlide = splitSlide;

function splitSlideCPS (slide: Slide) {
    return new Promise<Slide[]>((resolve) => {
        resolve(splitSlide(slide));
    });
}
(window as any).splitSlideCPS = splitSlideCPS;

function buildItem(item:Item): _H {
    /* Generate HTML from an Item
        This recurses with buildSlide as a Slide is an Item.
    */
    return match(item)
        .with({type: "html", h: select()}, (h) => h)
        .with({type: "figure", h: select()}, (h) => h)
        .with({type: "text", text: select()}, (t) => {
            let p = H("p");
            p.element.textContent = t;
            return p;
        })
        .with({items: __}, (s) => buildSlide(s))
        .exhaustive();
}

function buildItemCPS(item:Item) {
    return new Promise<_H>((resolve) => {
        resolve(buildItem(item));
    });
}

function buildSlide(slide:Slide): _H {
    /* Generate HTML from a Slide.
        All items in a Slide are built.
        Then, the HTML for the slide is generated based on slide type.
        This recurses with buildItem as a Slide is an Item.
    */
    return match(slide)
        .with({type: "title"}, () => {
            let section = H("section");
            let h1 = H("h1");
            return match(slide.items[0])
                .with({type: "text", text: select()}, (t) => {
                    h1.element.textContent = t;
                    section.append(h1);
                    return section;
                })
                .run();
        })
        .with({type: "title and content"}, () => {
            let section = H("section");
            let h1 = H("h1");
            return match(slide.items[0])
                .with({type: "text", text: select()}, (text) => {
                    h1.element.textContent = text;
                    section.append(h1);
                    for (const item of slide.items.slice(1)) {
                        section.append(buildItem(item));
                    }
                    return section;
                })
                .run();
        })
        .with({type: "figure and content"}, () => {
            let section = H("section");
            let table = H("table");
            let row = H("tr");
            let col1 = H("td");
            let col2 = H("td");

            col1.element.setAttribute("style", "width: 50%");

            section.append(table);
            table.append(row);
            row.append(col1);
            row.append(col2);

            col1.append(buildItem(slide.items[0]));
            for (const item of slide.items.slice(1)) {
                col2.append(buildItem(item));
            }
            return section;
        })
        .exhaustive();
}

function buildSlideCPS(slide:Slide) {
    return new Promise<_H>((resolve) => {
        resolve(buildSlide(slide));
    });
}

function buildPresentation(slidesElement){
    /* Generate HTML for an entire presentation.
        This is non-idempotent. The slidesElement passed in is modified.
        If a new slidesElement is returned, then the presentation library
        will have an reference to an out-of-date slide deck.
    */
        while (slidesElement.element.firstChild) {
            slidesElement.element.removeChild(slidesElement.element.firstChild);
        }
        for (const [index, slide] of presentation.slides.entries()) {

            let builtSlide = buildSlide(slide);
            slidesElement.append(
                builtSlide
            )
        }
}

(window as any).buildPresentation = buildPresentation;

function buildPresentationCPS(slidesElement) {
    return new Promise<void>((resolve) => {
        resolve(buildPresentation(slidesElement));
    });
}
(window as any).buildPresentationCPS = buildPresentationCPS;


function present(): void {
    /* Start the presentation.
        This function gets called when the page loads at the window level in the browser.
        It is responsible for setting up the presentation and building it.
        This function reads the source HTML and generates the ADT.
        buildPresentation is then called to generate the presentation.
    */

    // Structure created below is:
    // -----
    // <reveal>
    //   <slides>
    //     <section> -> 1 section per slide
    //   </slides>
    // </reveal>
    // -----

    let reveal = H("div")
        .addClass("reveal")

    let slides = H("div")
        .addClass("slides");

    (window as any).slides = slides;

    let peabody = H("body")
        .addClass("rst-content")
        .append(
            reveal
                .append(slides)
        )

    reveal.prepend(
        htmlToElement(
            `<style>
            h1, h2, h3, h4, h5, h6 {
                align: left;
                left: 0px;
                font-size:42px;
            }
            p {
                font-size: 22px;
                text-align: left;
            }
            .break{
                display:block;
                margin:0 0 1em;
            }
        </style>
        `
        )
    );

    // This codebase works for all documentation generated by Sphinx.
    // Most python projects use Sphinx, so this is a good starting point.
    // isOcamlsite is used to special case the element grabber and iterator
    // for the ocaml's documentation.
    let isOcamlsite = window.location.href.includes("ocaml.org");

    // This is a preorder traversal of the sections
    let webSections = Array.from(document.getElementsByClassName("section"));


    for (const [index, section] of webSections.entries()) {

        let slideTitle: string = section.children[0].textContent;

        if (isOcamlsite) {
            slideTitle = section.textContent;
        }

        if (index == 0) {
            // Add Main title slide
            presentation.slides.push({ type: "title", path: [section.id], items: [{ type: "text", text: slideTitle }] });
        }

        let isFigureAndContent = false;
        for (let [ii, child] of Array.from(isOcamlsite ? getNextSiblings(section) : section.children).entries()) {
            if (child.classList.contains("section")) {
                // The iterator iterates over all page contents, not just the sections.
                // We want to go through it section by section so that subsections are handled correctly.
                break;
            }

            if (ii == 0) {
                // Add Subsection title slide
                presentation.slides.push({ type: "title", path: [section.id], items: [{type: "text", text: slideTitle}] });
                continue;
            }

            let items: Item[] = [];
            if (child.tagName.toLowerCase() == "p") { // Paragraph
                let text = child.textContent;
                text = text.replace(/\n/g, " ");
                let paragraphs = splitParagraph(text);

                for(const p of paragraphs) {
                    // Create a text item with bullet points for each sentence
                    items.push({ type: "text", text: "- " + p });
                }

            }

            // If the next sibling is a figure, then we have a figure and caption
            if (child.classList.contains("image-reference")) {
                isFigureAndContent = true;
                presentation.slides.push({ type: "figure and content", path: [section.id], items: [{type: "figure", h: H(child)}] });
                continue;
            }

            let contents: Item[];
            if (items.length == 0) {
                // The element is not a paragraph or an image, so we don't touch the html.
                contents = [{type: "html", h: H(child) }];
            } else {
                contents = items;
            }

            if (isFigureAndContent) {
                // In a figure and content, we always want to see the figure.
                // So, we add content to the right column of the existing slide
                // instead of creating a new one.
                for (const item of contents) {
                    presentation.slides.at(-1).items.push(item);
                }
                continue;
            }

            contents.unshift({type: "text", text: slideTitle});

            // Base case: We have just have text and html so we create a title and content slide

            presentation.slides.push({ type: "title and content", path: [section.id], items: contents });

        }

    }

    buildPresentation(slides);

    // Download the CSS for the presentation library
    peabody.prepend(
        htmlToElement(
            `<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/reveal.js@4.1.2/dist/reveal.css">`
        )
    );

    peabody.append(
        // Download the JS for the presentation library
        createScriptSrcElement(
            `https://cdn.jsdelivr.net/npm/headjs@1.0.3/dist/1.0.0/head.min.js`,
            () => {
                peabody.append(
                    // Download the CSS for the presentation library
                    createScriptSrcElement(
                        `https://cdn.jsdelivr.net/npm/reveal.js@4.1.2/dist/reveal.min.js`,
                        () => {
                            peabody.append(
                                createScriptElement(
                                    `
                                setTimeout(()=>{
                                    window.initializePresentation(Reveal);
                                }, 1);
                               `
                                )
                            )
                        }
                    )
                )
            }
        )
    );

    // Inject the built presentation into the page
    let sherman = document.getElementsByTagName("body")[0];
    sherman.parentNode.replaceChild(peabody.element, sherman);

}
(window as any).present = present;

```

```
function initializePresentation(reveal: Reveal) {
    /* This function is called after the presentation has been built.
        It initializes the presentation library and sets up a callback for splitting slides.
    */

    // Initialize the presentation library
    // The values are the default values the library uses.
    Reveal.initialize({
        controls: true,
        width: '100%',
        height: '100%',
        progress: true,
        slideNumber: true,
        history: false,
        keyboard: true,
        overview: true,
        center: true,
        touch: true,
        loop: false,
        rtl: false,
        shuffle: false,
        fragments: true,
        embedded: false,
        help: true,
        showNotes: false,
        autoPlayMedia: null,
        autoSlide: 0,
        autoSlideStoppable: true,
        autoSlideMethod: Reveal.navigateNext,
        mouseWheel: false,
        hideAddressBar: true,
        previewLinks: true,
        transition: 'slide',
        transitionSpeed: 'default',
        backgroundTransition: 'fade',
        viewDistance: 3,
        parallaxBackgroundImage: '',
        parallaxBackgroundSize: '',
        parallaxBackgroundHorizontal: null,
        parallaxBackgroundVertical: null,
        display: 'block'
    });

    Reveal.on( 'slidetransitionend', event => {
        // This is called when the current slide changes.
        let slideIndex = event.indexh;
        let slideHeight = event.currentSlide.getBoundingClientRect().height;
        let windowHeight = window.innerHeight;

        if (slideHeight > windowHeight) {
            // Split the current slide if it is too tall for the browser window
            let slide = presentation.slides[slideIndex];
            let newSlides = splitSlide(slide);
            if (newSlides.length == 1) {
                return;
            }
            let [slide1, slide2] = newSlides;
            presentation.slides[slideIndex] = slide1;
            presentation.slides.splice(slideIndex, 0, slide2);

            // @ts-ignore
            buildPresentation(window.slides);

            // Force a refresh of the presentation
            Reveal.slide( event.indexh, event.indexv, event.indexf );
        }

    });
}
(window as any).initializePresentation = initializePresentation;


window.addEventListener("DOMContentLoaded", () => {
    /* Inject a "Present" button into the page so the user can launch the presentation. */
    let fries = document.getElementsByClassName("wy-breadcrumbs-aside")[0];

    let button = document.createElement("button");
    button.textContent = "Present!";
    button.onclick = present;

    fries.appendChild(button);
});
```

element.ts

```
"use strict";

export {H, Hwrappable, _H};

type Hwrappable = _H | HTMLElement | string | Element | Node

function H(element: Hwrappable) {
    if (element instanceof _H) return element
    if (element instanceof HTMLElement) return new _H(element)
    if (element instanceof Element) return new _H(element as HTMLElement)
    if (element instanceof Node) return new _H(element as HTMLElement)

    return new _H(document.createElement(element))
}

class _H {
    element: HTMLElement

    constructor(element: HTMLElement) {
        this.element = element
    }

    prepend(elements: Hwrappable | Hwrappable[]) {
        if (!Array.isArray(elements)) elements = [elements]
        this.element.prepend(...elements.map(e => H(e).element))
        return this
    }

    append(elements: Hwrappable | Hwrappable[]) {
        if (!Array.isArray(elements)) elements = [elements]
        this.element.append(...elements.map(e => H(e).element))
        return this
    }

    class(name: string = "") {
        this.element.className = name
        return this
    }

    addClass(name: string) {
        this.element.classList.add(name)
        return this
    }

    id(name: string) {
        this.element.id = name
        return this
    }

    copy() {
        return H(this.element.cloneNode(false) as HTMLElement)
    }

    deepcopy() {
        return H(this.element.cloneNode(true) as HTMLElement)
    }

}
```

split.ts

```typescript
export {splitParagraph, getToken};

import {Token, Tokenizer, StringIterator, TokenIterable} from 'lexing';
import { match, select } from 'ts-pattern';


const rules = [
    [/^$/, (m) => Token("EOF", null)],
    [/^\s+/, (m) => Token("Space", " ")],
    [/^\(.*?\)/, (m) => Token("Parenthesis", m[0])],
    [/^".*?"/, (m) => Token("Quote", m[0])],
    [/^'.*?'/, (m) => Token("Quote", m[0])],
    [/^(\.|\!|\?) /, (m) => Token("Punctuation", m[0])],
    [/^(\.|\!|\?)$/, (m) => Token("Punctuation", m[0])],
    // [/^\S+/, (match) => Token("Word", match[0])],
    [/^[^\s]+\.[^\s]+/, (m) => Token("Word", m[0])],
    [/^[^\.\!\?\s]+/, (m) => Token("Word", m[0])],
    [/^\.\.\./, (m) => Token("Word", m[0])],
    [/^(\.|\!|\?)[^\s]+/, (m) => Token("Word", m[0])],
];


// @ts-ignore
const tokenizer = new Tokenizer(rules);

function getToken(text: string) {
    /* Convert a string into a single token */
    const output = tokenizer.map(new StringIterator(text));
    return output.next();
}

function parse(output: TokenIterable<any>): string[] {
    /* Parse a token stream into a list of sentences. */
    let sentences = [];
    let sent = "";
    do {
        var token = output.next();
        sent +=
        match(token)
          .with({name: "EOF", value: select()}, (v) => "")
          .with({name: "Space", value: select()}, (v) => " ")
          .with({name: "Parenthesis", value: select()}, (v) => v)
          .with({name: "Quote", value: select()}, (v) => v)
          .with({name: "Punctuation", value: select()}, (v) => v[0])
          .with({name: "Word", value: select()}, (v) => v)
          .run();
        if (token.name == 'Punctuation' || (token.name == 'EOF' && sent.length > 0)) {
            sentences.push(sent);
            sent = "";
        }
    } while (token.name !== 'EOF');
    return sentences;
}

function splitParagraph(paragraph:string) {
    /* Split a paragraph into sentences. */
    const output = tokenizer.map(new StringIterator(paragraph));
    return parse(output);
}
```

utils.ts

```
"use strict";

export {htmlToElement, createScriptElement, createScriptSrcElement, getNextSiblings};

import {H, Hwrappable, _H} from './element';

function htmlToElement(html) {
    /* Create a HTMLElement from a string of HTML. */
    var template = document.createElement('template');
    html = html.trim(); // Never return a text node of whitespace as the result
    template.innerHTML = html;
    return H(<HTMLElement>template.content.firstChild);
}

function createScriptElement(code) {
    /* Create a script element from a string of JavaScript. */
    let s = document.createElement("script");
    s.type = "text/javascript";
    s.innerText = code;
    return s;
}

function createScriptSrcElement(src, onload) {
    /* Create a script element from a url pointing to a js file. */
    let s = document.createElement("script");
    s.type = "text/javascript";
    s.src = src;
    s.async = false;
    s.onload = onload;
    return s;
}

// https://stackoverflow.com/questions/4378784/how-to-find-all-siblings-of-the-currently-selected-dom-object
function getNextSiblings(elem) {
    /* Get all siblings following an element (elem). */
    var sibs = [];
    while (elem = elem.nextElementSibling) {
        if (elem.nodeType === 3) continue; // text node
        sibs.push(elem);
    }
    return sibs;
}
```

present.test.ts

```typescript
import { describe } from 'mocha';
import { expect } from 'chai';
import { present, buildPresentation, buildPresentationCPS, splitSlide, splitSlideCPS, buildItem, buildItemCPS, buildSlide, buildSlideCPS, Slide, Item, Text, Html, Figure, Presentation } from '../src/present';
import { H } from '../src/element';

describe("splitSlide", () => {

    describe("title", () => {
        it("size small slide", () => {
            let slide: Slide = {type: "title", items: [], path: []};
            let splitSlides = splitSlide(slide);
            expect(splitSlides.length).to.equal(1);
        });

        it("size big slide", () => {
            let slide: Slide = {type: "title", items: [{type: "text", text: "asdf1"}, {type: "text", text: "asdf2"}], path: []};
            let splitSlides = splitSlide(slide);
            expect(splitSlides.length).to.equal(1);
        });

        it("contents small slide", () => {
            let slide: Slide = {type: "title", items: [], path: []};
            let splitSlides = splitSlide(slide);
            expect(splitSlides[0]).to.deep.equal(slide);
        });

        it("contents big slide", () => {
            let slide: Slide = {type: "title", items: [{type: "text", text: "asdf1"}, {type: "text", text: "asdf2"}, {type: "text", text: "asdf3"}], path: []};
            let splitSlides = splitSlide(slide);
            expect(splitSlides[0]).to.deep.equal(slide);
        });

    });

    describe("title and contents", () => {
        it("size small slide", () => {
            let slide: Slide = {type: "title and content", items: [{type: "text", text: "asdf1"}, {type: "text", text: "asdf2"}], path: []};
            let splitSlides = splitSlide(slide);
            expect(splitSlides.length).to.equal(1);
        });

        it("size big slide", () => {
            let slide: Slide = {type: "title and content", items: [{type: "text", text: "asdf1"}, {type: "text", text: "asdf2"}, {type: "text", text: "asdf3"}], path: []};
            let splitSlides = splitSlide(slide);
            expect(splitSlides.length).to.equal(2);
        });

        it("contents small slide", () => {
            let slide: Slide = {type: "title and content", items: [{type: "text", text: "asdf1"}, {type: "text", text: "asdf2"}], path: []};
            let splitSlides = splitSlide(slide);
            expect(splitSlides[0]).to.deep.equal(slide);
        });

        it("contents big slide", () => {
            let slide: Slide = {type: "title and content", items: [{type: "text", text: "asdf1"}, {type: "text", text: "asdf2"}, {type: "text", text: "asdf3"}], path: []};
            let splitSlides = splitSlide(slide);
            expect(splitSlides[0]).to.deep.equal({type: "title and content", items: [{type: "text", text: "asdf1"}, {type: "text", text: "asdf2"}], path: []});
            expect(splitSlides[1]).to.deep.equal({type: "title and content", items: [{type: "text", text: "asdf1"}, {type: "text", text: "asdf3"}], path: []});
        });

    });


    describe("figure and contents", () => {
        it("size small slide", () => {
            let slide: Slide = {type: "figure and content", items: [{type: "figure", h: H("q")}, {type: "text", text: "asdf2"}], path: []};
            let splitSlides = splitSlide(slide);
            expect(splitSlides.length).to.equal(1);
        });

        it("size big slide", () => {
            let slide: Slide = {type: "figure and content", items: [{type: "figure", h: H("q")}, {type: "text", text: "asdf2"}, {type: "text", text: "asdf3"}], path: []};
            let splitSlides = splitSlide(slide);
            expect(splitSlides.length).to.equal(2);
        });

        it("contents small slide", () => {
            let slide: Slide = {type: "figure and content", items: [{type: "figure", h: H("q")}, {type: "text", text: "asdf2"}], path: []};
            let splitSlides = splitSlide(slide);
            expect(splitSlides[0]).to.deep.equal(slide);
        });

        it("contents big slide", () => {
            let slide: Slide = {type: "figure and content", items: [{type: "figure", h: H("q")}, {type: "text", text: "asdf2"}, {type: "text", text: "asdf3"}], path: []};
            let splitSlides = splitSlide(slide);
            expect(splitSlides[0].type).to.equal("figure and content");
            expect(splitSlides[0].items[0].type).to.equal("figure");
            // @ts-ignore
            expect(splitSlides[0].items[0].h.element.tagName).to.equal("Q");
            expect(splitSlides[0].items[1].type).to.equal("text");
            // @ts-ignore
            expect(splitSlides[0].items[1].text).to.equal("asdf2");
            expect(splitSlides[0].items.length).to.equal(2);


            expect(splitSlides[1].type).to.equal("figure and content");
            expect(splitSlides[1].items[0].type).to.equal("figure");
            // @ts-ignore
            expect(splitSlides[1].items[0].h.element.tagName).to.equal("Q");
            expect(splitSlides[1].items[1].type).to.equal("text");
            // @ts-ignore
            expect(splitSlides[1].items[1].text).to.equal("asdf3");
            expect(splitSlides[1].items.length).to.equal(2);
        });

    });

});


describe("buildItem", () => {
    it("html", () => {
        let item: Item = {type: "html", h: H("p")};
        let html = buildItem(item);
        expect(html.element.outerHTML).to.deep.equal("<p></p>");
    });

    it("text", () => {
        let item: Item = {type: "text", text: "asdf"};
        let html = buildItem(item);
        expect(html.element.outerHTML).to.equal("<p>asdf</p>");
    });

    it("figure", () => {
        let item: Item = {type: "figure", h: H("img")};
        item.h.element.setAttribute("src", "asdf");
        let html = buildItem(item);
        expect(html.element.outerHTML).to.equal("<img src=\"asdf\">");
    });

    it("slide", () => {
        let item: Slide = {type: "title", items: [{type: "text", text: "asdf"}], path: []};
        let html = buildItem(item);
        expect(html.element.outerHTML).to.equal("<section><h1>asdf</h1></section>");
    });
});


describe("buildSlide", () => {
    it("title", () => {
        let slide: Slide = {type: "title", items: [{type: "text", text: "asdf"}], path: []};
        let html = buildSlide(slide);
        expect(html.element.outerHTML).to.equal("<section><h1>asdf</h1></section>");
    });

    it("title and content", () => {
        let slide: Slide = {type: "title and content", items: [{type: "text", text: "asdf1"},
{type: "text", text: "asdf2"}], path: []};
        let html = buildSlide(slide);
        expect(html.element.outerHTML).to.equal("<section><h1>asdf1</h1><p>asdf2</p>
</section>");
    });

    it("figure and content", () => {
        let slide: Slide = {type: "figure and content", items: [{type: "figure", h: H("q")},
{type: "text", text: "asdf2"}], path: []};
        let html = buildSlide(slide);
        expect(html.element.outerHTML).to.equal("<section><table><tr><td style=\"width: 50%\">
<q></q></td><td><p>asdf2</p></td></tr></table></section>");
    });
```

```
});
```

element.test.ts

```
import { describe } from 'mocha';
import { expect } from 'chai';
import {H, Hwrappable, _H} from '../src/element';

describe("H", () => {
    it("_H", () => {
        let h: _H = H("a");
        expect(H(h)).to.equal(h);
    });
    it("HTMLElement", () => {
        let h: HTMLElement = H("a").element;
        expect(H(h).element).to.equal(h);
    });
    it("Element", () => {
        let h: Element = H("a").element;
        expect(H(h).element).to.equal(h as HTMLElement);
    });
    it("Node", () => {
        let h: Node = H("a").element.cloneNode();
        expect(H(h).element).to.equal(h as HTMLElement);
    })
});

describe("prepend", () => {
    it("Hwrappable", () => {
        let h: _H = H("a");
        let h2: _H = H("b");
        h.prepend(h2);
        expect(h.element.firstChild).to.equal(h2.element);
    });
    it("Hwrappable[]", () => {
        let h: _H = H("a");
        let h2: _H = H("b");
        h.prepend([h2]);
        expect(h.element.firstChild).to.equal(h2.element);
    });
});

describe("append", () => {
    it("Hwrappable", () => {
        let h: _H = H("a");
        let h2: _H = H("b");
        h.append(h2);
        expect(h.element.lastChild).to.equal(h2.element);
    });
    it("Hwrappable[]", () => {
        let h: _H = H("a");
        let h2: _H = H("b");
        h.append([h2]);
        expect(h.element.lastChild).to.equal(h2.element);
    });
});

describe("class", () => {
    it("name", () => {
        let h: _H = H("a");
        h.class("a");
        expect(h.element.className).to.equal("a");
    });
});

describe("id", () => {
    it("name", () => {
        let h: _H = H("a");
        h.id("a");
        expect(h.element.id).to.equal("a");
    });
});

describe("addClass", () => {
    it("name", () => {
        let h: _H = H("a");
        h.addClass("a");
        expect(h.element.classList.contains("a")).to.be.true;
    });
});

describe("copy", () => {
    it("copy", () => {
        let h: _H = H("a");
        let h2: _H = h.copy();
        expect(h2.element.outerHTML).to.equal(h.element.outerHTML);
    });

    it("deepcopy", () => {
        let h: _H = H("a");
        let h2: _H = h.deepcopy();
        expect(h2.element.outerHTML).to.equal(h.element.outerHTML);
    });

})
```

```
import { describe } from 'mocha';
import { expect } from 'chai';
import { getToken, splitParagraph } from '../src/split';

describe("splitParagraph", () => {
    it("newlines", () => {
        let text = "qwe\nqwe\nqwe";
        let result = splitParagraph(text);
        expect(result).to.deep.equal(["qwe qwe qwe"]);
    });

    it("spaces", () => {
        let text = "qwe qwe qwe";
        let result = splitParagraph(text);
        expect(result).to.deep.equal(["qwe qwe qwe"]);
    });

    it("no quotes and no parenthesis", () => {
        let text = "qwe qwe. hello. qwe";
        let result = splitParagraph(text);
        expect(result).to.deep.equal(["qwe qwe.", "hello.", "qwe"]);
    });

    it("quotes", () => {
        let text = "qwe. \"qwe. hello. \". qwe";
        let result = splitParagraph(text);
        expect(result).to.deep.equal(["qwe.", "\"qwe. hello. \".", "qwe"]);
    });

    it("parenthesis", () => {
        let text = "qwe. (qwe. hello. ). qwe";
        let result = splitParagraph(text);
        expect(result).to.deep.equal(["qwe.", "(qwe. hello. ).", "qwe"]);
    })

    it("real world 1", () => {
        let text = "PWM motor controllers can be controlled in the same way as a CAN motor controller. For a more detailed background on how they work, see PWM Motor Controllers in Depth. To use a PWM motor controller, simply use the appropriate motor controller class provided by WPI and supply it the port the motor controller(s) are plugged into on the roboRIO. All approved motor controllers have WPI classes provided for them.";
        let result = splitParagraph(text);
        expect(result).to.deep.equal([
            "PWM motor controllers can be controlled in the same way as a CAN motor controller.",
            "For a more detailed background on how they work, see PWM Motor Controllers in Depth.",
            "To use a PWM motor controller, simply use the appropriate motor controller class provided by WPI and supply it the port the motor controller(s) are plugged into on the roboRIO.",
            "All approved motor controllers have WPI classes provided for them."
        ]);
    });

    it("real world 2", () => {
        let text = "Under the interactive system, the user types OCaml phrases terminated by ;; in response to the # prompt, and the system compiles them on the fly, executes them, and prints
the outcome of evaluation. Phrases are either simple expressions, or let definitions of identifiers (either values or functions).";
        let result = splitParagraph(text);
        expect(result).to.deep.equal([
            "Under the interactive system, the user types OCaml phrases terminated by ;; in response to the # prompt, and the system compiles them on the fly, executes them, and prints the outcome of evaluation.",
            "Phrases are either simple expressions, or let definitions of identifiers (either values or functions)."
        ]);
    });

});


describe("tokenizer", () => {
    it("eof", () => { expect(getToken("").name).to.equal("EOF");});
    it("space", () => { expect(getToken(" ").name).to.equal("Space");});
    it("space", () => { expect(getToken("          ").name).to.equal("Space");});
    it("space", () => { expect(getToken("\n").name).to.equal("Space");});
    it("parenthesis", () => { expect(getToken("()").name).to.equal("Parenthesis");});
    it("parenthesis", () => { expect(getToken("(qwe. asd. 1.23
)").name).to.equal("Parenthesis");});
    it("quote", () => { expect(getToken("\"qwe. asd. 1.23 \"").name).to.equal("Quote");});
    it("quote", () => { expect(getToken("\'qwe. asd. 1.23 \'").name).to.equal("Quote");});
    it("punctuation", () => { expect(getToken(".").name).to.equal("Punctuation");});
    it("punctuation", () => { expect(getToken("?").name).to.equal("Punctuation");});
    it("punctuation", () => { expect(getToken("!").name).to.equal("Punctuation");});
    it("punctuation", () => { expect(getToken(". ").name).to.equal("Punctuation");});
    it("punctuation", () => { expect(getToken("? ").name).to.equal("Punctuation");});
    it("punctuation", () => { expect(getToken("! ").name).to.equal("Punctuation");});
    it("word", () => { expect(getToken("hello").name).to.equal("Word");});
    it("word", () => { expect(getToken("1.23").name).to.equal("Word");});
    it("word", () => { expect(getToken(".toggle()").name).to.equal("Word");});
    it("word", () => { expect(getToken("...").name).to.equal("Word");});
});
```

utils.test.ts

```
import { describe } from 'mocha';
import { expect } from 'chai';
import { htmlToElement, createScriptElement, createScriptSrcElement, getNextSiblings } from
'../src/utils';

describe("htmlToElement", () => {
    describe("test element creation", () => {
        it("div", () => {
            let el = htmlToElement("<div>");
            expect(el.element.tagName).to.equal("DIV");
        });

        it("h1", () => {
            let el = htmlToElement("<h1>qwe</h1>");
            expect(el.element.tagName).to.equal("H1");
        });
    });

    describe("test element attributes", () => {
        it("id", () => {
            let el = htmlToElement("<div id='asd'>");
            expect(el.element.id).to.equal("asd");
        });

        it("class", () => {
            let el = htmlToElement("<div class='asd'>");
            expect(el.element.className).to.equal("asd");
        });

    });

    describe("test element children", () => {

        it("element", () => {
            let el = htmlToElement("<div><h1>qwe</h1></div>");
            expect(el.element.children[0].tagName).to.equal("H1");
        });

    });

});

describe("createScriptElement", () => {
    it("element", () => {
        let el = createScriptElement("qwe");
        expect(el.tagName).to.equal("SCRIPT");
    })

    it("type", () => {
        let el = createScriptElement("qwe");
        expect(el.type).to.equal("text/javascript");
    });

    it("code", () => {
        let el = createScriptElement("qwe");
        expect(el.innerText).to.equal("qwe");
    });

});

describe("createScriptSrcElement", () => {
    it("element", () => {
        let el = createScriptSrcElement("qwe", () => {});
        expect(el.tagName).to.equal("SCRIPT");
    })

    it("type", () => {
        let el = createScriptSrcElement("qwe", () => {});
        expect(el.type).to.equal("text/javascript");
    });

    it("src", () => {
        let el = createScriptSrcElement("qwe", () => {});
        expect(el.src).to.equal("qwe");
    });

});

describe("getNextSiblings", () => {
    it("0", () => {
        let el = htmlToElement("<div><h1>qwe</h1></div>");
        expect(getNextSiblings(el.element).length).to.equal(0);
    });

    it("1", () => {
        let el = htmlToElement("<div><h1>qwe</h1><h2>qwe</h2></div>");
        expect(getNextSiblings(el.element.children[0]).length).to.equal(1);
    });
});
```

endtoend.test.ts

```typescript
import { describe } from 'mocha';
import { expect } from 'chai';
import { Builder, By, Key } from 'selenium-webdriver';
import {readFileSync } from 'fs';


describe("end to end", () => {
    it("test", async () => {
        const driver = new Builder().forBrowser('chrome').build();
        const file = readFileSync("build/present.js", "utf8");
        await driver.get("https://docs.wpilib.org/en/stable/docs/controls-overviews/control-system-hardware.html");
        await driver.sleep(1000);
        await driver.executeScript(file);
        await driver.executeScript(`present();`);
        await driver.sleep(2000);
        let button = await driver.findElement(By.className("navigate-right enabled"));
        for (let i = 0; i < 12; i++) {
            await driver.sleep(500);
            // @ts-ignore
            await button.click();
        }
        await driver.quit();
    });
});
```