卒業論文

ユーザーメモソフト my_help の GUI 版となる org_viewer の開発

関西学院大学理工学部

情報科学科 西谷研究室

27017580 谷口旭人

2022 年 3 月

概 要

西谷研では，メモを取るための CLI ソフトウェアである my_help を利用している．しかし，

- GUI でのボタンやスクロールバーなどの直感的な操作で動かすことができない
- キーボードショートカットを利用する必要がある
- リスト表示やソースコードのカラーライズ，太字のようなリッチな表示をすることが出来ない

といった，初学者には扱いづらい点があった．そこで，上記の点を改善すべく，本研究では my_help の GUI 版となる org_viewer の開発を行なった．

org viewer はベースとして Electron を用いて開発し、React や Bootstrap を用いて、以下の点を取り入れたアプリケーションを開発した。

- ファイルの一覧を階層構造で表示する
- ファイルの作成や編集，削除等が GUI で操作することができる
- 作成したファイルをリッチな表示で確認することができる

目 次

第 1 章　序論 **3**

第 2 章　手法 **5**

2.1 Electron 5

2.2 react 5

2.3 Electron + react の使い方 5

2.3.1 初期準備 5

2.3.2 各種ファイル作成 6

2.3.3 各種設定 6

2.3.4 最低限のコード記述 8

2.3.5 動作確認 11

2.4 bootstrap 11

2.5 pandoc 11

2.5.1 pandoc とは 11

2.5.2 使用方法 12

第 3 章　結果 **13**

3.1 ディレクトリ画面 13

3.1.1 ファイルを新規作成する 14

3.1.2 ファイルを削除する 15

3.1.3 ファイルを編集する 15

3.1.4 ファイルの中身を表示する 15

3.2 ビュワー画面 15

3.2.1 html を解析し，タイトルを抽出・リンクを作成する 17

3.2.2 ビュワー画面に表示する 17

3.2.3 コードブロックについて 18

第4章 結論 **20**

4.1 今後の課題 20

4.1.1 学習効率の計測 20

4.1.2 テストの作成 20

付録A webpackのソースコード **23**

付録B hタグを抽出するソースコード **28**

図 目 次

3.1 起動時に表示されるディレクトリ画面の図. 13
3.2 新規作成時のファイル名入力画面の図. 14
3.3 ファイルを選択して表示するときのビュワー画面の図. 16
3.4 Prismjs を使用しない場合 (左) と，使用した場合 (右) 19

第1章　序論

西谷研においては，メモを取るための Command Line Interface(CLI) ソフトである my_help を利用している．これは，emacs と CLI を利用することで，ターミナル上でのメモを取ることができるソフトである．

my_help とは，CUI(CLA) ヘルプの Usage 出力を真似て，user 独自の help を作成・提供する gem である．特徴としては,

- user が自分にあった man を作成
- 雛形を提供
- おなじ format, looks, 操作, 階層構造
- すぐに手が届く
- それらを追加・修正・削除できる

の 5 点があげられている [1].

CLI ソフトは，GUI でのボタンやスクロールバーなどの直感的な操作で動かすことができず，キーボードショートカットを利用する必要があるため，慣れる必要があること，また作成したメモをリスト表示やソースコードのカラーライズ，太字のようなリッチな表示をすることが出来ない．そのため，初学者には CLI のみで扱うことは難しい．そこで，本研究では，my_help の Graphical User Interface(GUI) 版となる org_viewer を開発することを目的とした．

my_help の編集やフォーマットには，emacs の org mode を利用している．org-mode を用いて資料などを作成・共有するための，表示プログラムが存在しているか調べた．orgro[2] というスマートフォン向けアプリケーションや，org-web[3] というブラウザ向けアプリケーションが存在していたが，いずれも読み取り専用や，ローカル環境と共有することが難しいなど，目標に合致するプログラムを発見することは出来なかった．

org_viewer は，my_help の GUI 版となるアプリケーションを目指している．特徴として，

- ファイルの作成や編集，削除等が GUI で操作することができる
- 作成したファイルをリッチな表示で確認することができる

を持つことを開発の目標とした．

第2章　手法

2.1　Electron

Electron[4] はGithub が開発，管理している，オープンソースなソフトウェアフレームワークである．ランタイムのNode.js とChronium を組み合わせ，web 技術を用いてデスクトップのGUI アプリケーションを開発することができる．また，パッケージングも行うことができ，アプリケーションとして配布することが可能である．本研究では，Electron を基盤として，アプリケーションを開発していく．

2.2　react

React[6] は，Meta とコミュニティによって開発されている，ユーザインタフェース構築のためのJavaScript ライブラリである．これは，JavaScript のコンポーネントを定義することで，簡単にDOM を操作することができる．

2.3　Electron ＋ react の使い方

以下は，エンジニアコミュニティサービスであるqiita に投稿した記事/cite{qiita} を，本論文用に改変したものである．

2.3.1　初期準備

Electron はNode.js をベースに起動するため，’npm’ コマンドを用いてnode の初期設定を行う．‘¿ npm init -y‘

また，本研究で作成したorg_viewer を作成する際は，javascript を拡張したtypescript を用いたため，それを含めて必要なパッケージをインストールする．

```
% npm install --save-dev electron typescript ts-node @types/node @types/react @types/r
% npm install -save react react-dom
```

また，typescript はコンパイルが必要なため，それらを webpack で管理する．そのため，webpack と，コンパイル時に役に立ついくつかのライブラリもインストールする．

```
% npm install --save-dev webpack webpack-cli ts-loader css-loader mini-css-extract-plu
% npm install --save-dev rimraf cross-env npm-run-all
```

2.3.2　各種ファイル作成

Electron はメインプロセス，レンダラープロセス，及びそれらを繋ぐプレロードの３種類がある．そのため，それぞれのファイルを作成する必要がある．本研究では，src ディレクトリ及び以下のファイルを作成した．

- ‘src/main.ts‘(メインプロセス)
- ‘src/index.html‘(レンダラープロセスの html 部分)
- ‘src/preload.ts‘(プレロード)
- ‘src/renderer.tsx‘(レンダラープロセス)

2.3.3　各種設定

1. package.json

   node.js は，package.json 内に記述された”scripts” 内にコマンドを記述することによって，頻繁に使用するコマンドを，”npm run”で実行することができる．そのため，よく使用する実行コマンドを追加しておく．また”main” を”dist/main.js” に変更しておく．

   ```
   "scripts": {
     "start": "run-s clean build serve",
   ```

```
  "clean": "rimraf dist",
  "build": "cross-env NODE_ENV=\"development\" webpack --progress",
  "serve": "electron ."
},
"main": "dist/main.js",
```

先ほど作成した scripts は，それぞれ以下の内容となっている．

- clean：dist ディレクトリとその中身を削除するコマンド
- build：webpack を使用してコンパイルするコマンド
- serve：Electron を起動するコマンド
- start：上記３つを順番に実行するコマンド

2. Typescript

typescript を使用するには，tsconfig.json を作成する必要がある．これは，以下のコマンドで生成することが出来る．

```
% npx tsc --init
```

これで tsconfig.json が出来たら，中身を少し変更する．以下は，該当する行のみを記載している．

```
"target": "es2020",
"lib": ["DOM", "ES2020"],
"jsx": "react-jsx",
"sourceMap": true,
"outDir": "./dist",
```

3. Webpack

typescript を javascript にコンパイルしたり，複数のファイルを１つにまとめたりするために，webpack を使用する．webpack.config.ts ファイルを生成し，以下の記述をすることで，webpack の設定を行うことができる．

webpack はコンパイラのような役割をし，変化するコードではないため，付録に記述した．

2.3.4 最低限のコード記述

Electron として最低限動作するコードを記述していく．

1. main.ts

   ベースとなる main.ts でレンダラープロセスを生成する必要があるため，以下を記述する．

```
import path from 'path';
import { app, BrowserWindow } from 'electron';

/**
* BrowserWindow インスタンスを作成する関数
*/
const createWindow = () => {
  const mainWindow = new BrowserWindow({
    webPreferences: {
      nodeIntegration: false,
      contextIsolation: true,
      preload: path.join(__dirname, 'preload.js'),
    },
  });

  // 開発時にはデベロッパーツールを開く
  if (process.env.NODE_ENV === 'development') {
    mainWindow.webContents.openDevTools({ mode: 'detach' });
  }
```

```
  // レンダラープロセスをロード
  mainWindow.loadFile('dist/index.html');
};

/**
* アプリを起動する準備が完了したら BrowserWindow インスタンスを作成し，
* レンダラープロセス（index.html とそこから呼ばれるスクリプト）を
* ロードする
*/
app.whenReady().then(async () => {
  // BrowserWindow インスタンスを作成
  createWindow();
});

// すべてのウィンドウが閉じられたらアプリを終了する
app.once('window-all-closed', () => app.quit());
```

2. index.html

GUI アプリケーションとして，ユーザーが操作する部分は index.html である．中身は renderer.tsx で操作するため，必要最低限の html となる．

```
<!DOCTYPE html>
<html lang="ja">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <!-- CSP の設定 -->
    <meta http-equiv="Content-Security-Policy" content="default-src 'self';" />
```

```html
    <title>Electron Title</title>
  </head>
  <body>
    <!-- react コンポーネントのマウントポイント -->
    <div id="root"></div>
  </body>
</html>
```

3. preload.ts

   最低限の動作上ではメインとレンダラーで通信を行わないが，メインプロセスで呼び出すため，空ファイルを置いておく．

4. renderer.tsx

   先程の index.html を変更するコードを記述する．ここでは React を使っている．

```tsx
import React from "react";
import ReactDOM from "react-dom";

class RootDiv extends React.Component<{}, {}> {
  constructor(props: {}) {
    super(props);
  }
  render = () => {
    return <>Hello World</>;
  };
}

ReactDOM.render(<RootDiv />, document.getElementById("root"));
```

2.3.5　動作確認

以下のコマンドを実行することで動作を確認することができる．

```
% npm run start
```

今回はウィンドウが表示され,「Hello World」が表示されれば成功した．これで，最低限 Electron + React では動作する．

2.4　bootstrap

bootstrap[7] とは，twitter 開発者によって作成された，ウェブサイトや web アプリケーションを作成する際に使用するフロントエンドのフレームワークである．html には css を使用して，Dom 要素をデザインすることが出来る．このフレームワークは，ブラウザのレイアウトを作成するために必要な css の知識がなくても，手軽に高度なレイアウトを作成できる．本研究では，GUI 部分が web アプリケーションと同じような構造で生成できるため，このフレームワークを使用している．

2.5　pandoc

my_help は org mode で作成するため，そのままでは書かれた文章を読むには適さない．そこで，文章を編集する際は org mode で書き込み，書いた内容を確認したい時は表示に特化した形式に変換して表示する．そうすることにより，書き込んだ内容を簡単に振り返ることが可能になる．本研究では，表示に特化した形式として，html を選択した．

2.5.1　pandoc とは

pandoc[8] は，あるマークアップ形式を別の形式に変換を行うためのツールである．かなり多様な形式の変換が可能であり，org mode と html を変換することも可能である．また，オプションも多数存在しているため，変換する際に任意の出力を出すことが可能である．

2.5.2 使用方法

pandoc は homebrew でインストールすることが出来る．

```
% brew install pandoc
```

その後，org mode で書き込んだ text.org を html に変換した場合は以下のコマンドを実行する．

```
% pandoc text.org -o text.html
```

第3章　結果

3.1　ディレクトリ画面

本アプリケーションが起動した際に，最初に表示される．my_help では，コマンド”my_help list”を実行すると，”~/.my_help”に保存されているファイルの一覧が表示されるが，それと同じく”~/.my_help”のディレクトリが表示される．

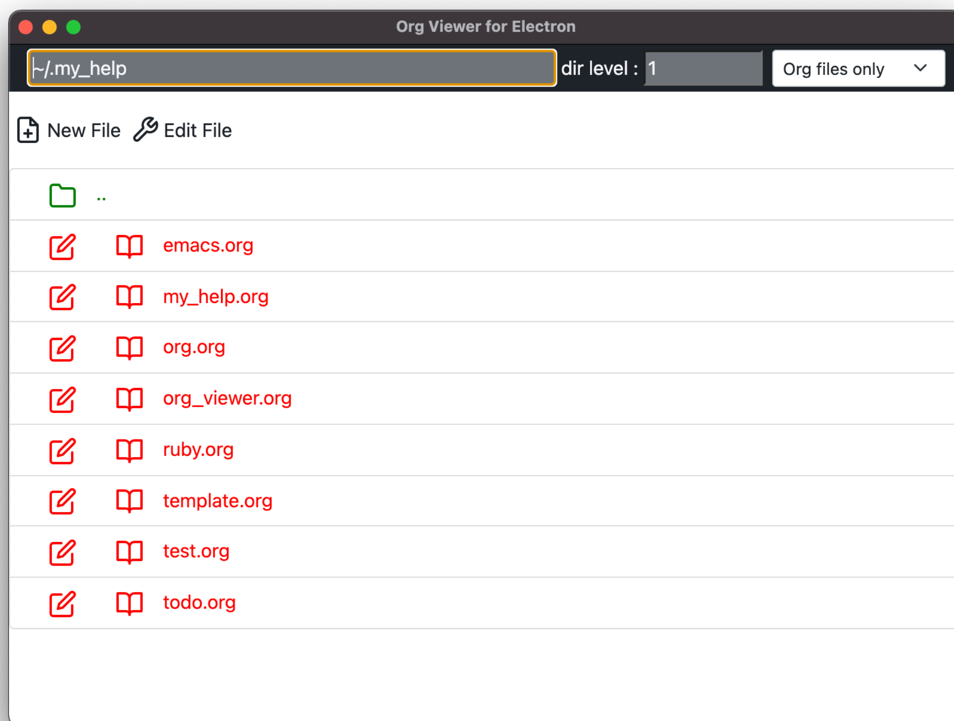

図 3.1: 起動時に表示されるディレクトリ画面の図.

この画面では，以下の操作が可能である．

- ファイルを新規作成する

- ファイルを削除する
- ファイルを編集する
- ファイルの中身を表示する

3.1.1 ファイルを新規作成する

my_help では，コマンド"my_help new ファイル名"を実行すると，"~/.my_help"以下に，該当のファイル名が新規で作成される．それに倣い，"New File"を選択すると，新しくファイルが作成されるようにした．

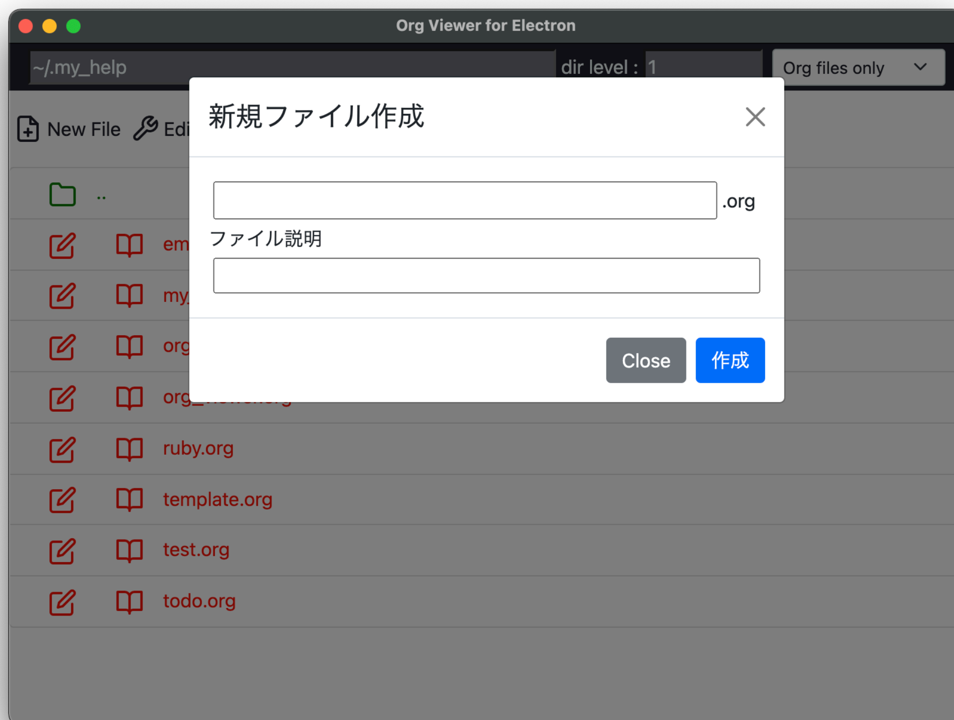

図 3.2: 新規作成時のファイル名入力画面の図.

New File を選択した時にポップアップが表示され，ここにはファイル名と，ファイル説明を記入することが出来る．ファイル名は，my_help の引数と同等である．ファイル説明は，my_help で新規作成される場合，テンプレートがデフォルトで記入されているが，

その中にあるファイル説明の項目を，ファイル作成時に書き入れることが出来るように改良したものである．

3.1.2　ファイルを削除する

my_help では，コマンド”my_help delete ファイル名”を実行すると，”~/.my_help”以下にある，該当のファイルが削除される．それに倣い，”Edit File”を選択すると，削除するアイコンが表示され，該当のファイルを選択すると削除されるようにした．また，誤って削除しないよう，削除する前に一度ポップアップが表示され，意図せず削除するのを防止している．

3.1.3　ファイルを編集する

my_help では，コマンド”my_help edit ファイル名”を実行すると，”~/.my_help”以下にある，該当のファイルを編集することができる．それに倣い，編集ボタンを選択すると，編集することが出来るようにした．実際に編集するのはアプリケーション内ではなく，テキストエディタである Emacs の GUI 版アプリケーションを利用する．

3.1.4　ファイルの中身を表示する

メモとして記述するのは，org mode で記述する．これは，マークアップ言語の一種であり，プレーンテキストと意味を持つ記号で構成されている．そのため，文章を構成するためには簡単に記述することが出来るが，見返す際は，記号を認識して読む必要がある．そのため，より見やすくするように，内部で形式を変更して表示するようにした．これにより，記号を認識して読む必要がなく，簡単に見返すことが可能になる．

3.2　ビュワー画面

ビュワー画面は，ファイルを表示する際に表示される画面である．

このビュアーは，ファイルをそのまま表示するのではなく，読みやすいように変換して表示している．内部では，以下のように動作している．

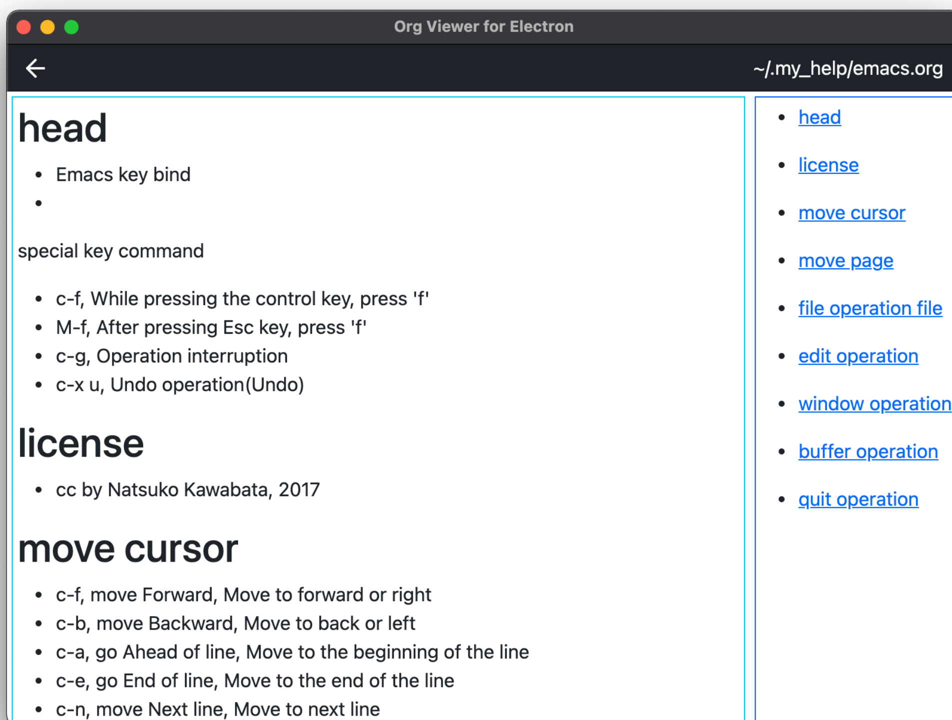

図 3.3: ファイルを選択して表示するときのビュワー画面の図.

- pandocを利用して，org modeの文章をhtmlに変換する
- htmlを解析し，タイトルを抽出・リンクを作成する
- ビュワー画面に表示する

以下に詳しい動作を示す.

* pandocを利用して，org modeの文章をhtmlに変換する

node.jsは，childprocessというモジュールを利用して，シェルコマンドを実行することが出来る．これを利用して，pandocを利用して，org modeの文章をhtmlに変換する.

pandocは，以下のコマンドで実行することが出来る.

```
% pandoc -f org -t html ファイル名.org
```

-oのオプションで，出力先のファイルを指定することができるが，このコマンドでは，出力先のファイルを指定していない．これにより，変換されたHTML文がそのまま出力される．これを利用し，特定のファイルを作成することなく，node.jsの変数として保持することが出来る.

3.2.1 htmlを解析し，タイトルを抽出・リンクを作成する

先ほど取得したhtmlは，そのまま表示することが出来るが，分量が大きくなると，目的の文章を探すのに時間がかかってしまう．そのため，htmlを解析し，タイトルとなるh1からh3タグを取得する.

hタグを取得する関数は，付録に記載している.

この関数に，DOMオブジェクトとなるhtmlを引数として渡すことで，タイトルだけを一覧にしたもう一つのhtmlを取得することができる.

3.2.2 ビュワー画面に表示する

先ほどの2つの変数から，図3.3のように，左側に本文，右側に目次を表示する．また，右側の目次はクリックすることができ，クリックするとそのタイトルの位置までスクロールする．これにより，欲しい情報のところまで一度に移動することができる.

3.2.3 コードブロックについて

pandoc で変換した html は，css がついていない為，コードブロックも他の文字と同じ色で表示される．その為，css を利用して，コードブロックに色をつけることにした．本アプリケーションでは，Prismjs[9] というライブラリを利用して，コードブロックに色をつける．これにより，コードブロックを読みやすくすることが出来る．

しかし，pandoc で生成した html の DOM オブジェクトは，Prismjs が指定する記述ではなく，そのままでは色をつけられない．

例えば，文章内で以下の javascript のコードが記述されているとする．

```
const obj = {
  val1: 200,
  val2: 400,
};


console.log(obj.val1);
```

これを pandoc で変換すると，以下のような HTML が出力される．

```
<div class="sourceCode" id="cb1">
  <pre class="sourceCode javascript">
    <code class="sourceCode javascript">
      <span id="cb1-1"><a href="#cb1-1" aria-hidden="true" tabindex="-1"></a><span cla
      <span id="cb1-2"><a href="#cb1-2" aria-hidden="true" tabindex="-1"></a>  <span c
      <span id="cb1-3"><a href="#cb1-3" aria-hidden="true" tabindex="-1"></a>  <span c
      <span id="cb1-4"><a href="#cb1-4" aria-hidden="true" tabindex="-1"></a>}<span cl
      <span id="cb1-5"><a href="#cb1-5" aria-hidden="true" tabindex="-1"></a></span>
      <span id="cb1-6"><a href="#cb1-6" aria-hidden="true" tabindex="-1"></a><span cla
    </code>
  </pre>
</div>
```

Prismjs は code ブロックのクラス名に”language-xxx” (xxx は言語) という名称が必要で，なおかつ中身はシンプルなコードのみである必要がある．

その為，pandoc で生成した html の DOM オブジェクトを，Prismjs に対応するようにする．

pandoc の引数には，フィルターをかけるようにするオプションが存在しているため，これを利用する．

フィルターは，pandoc-filters[10] というライブラリが存在してたため，これを利用する．pandoc には，–lua-filter という引数を追加することで，フィルターを設定することが出来る．

```
% pandoc --lua-filter "フィルターのパス" -f org -t html "org ファイルのパス"
```

今回は先程の pandoc-filters の中にある，”standard-code” をお借りした．

これで，Prismjs に対応できる html を生成できた．

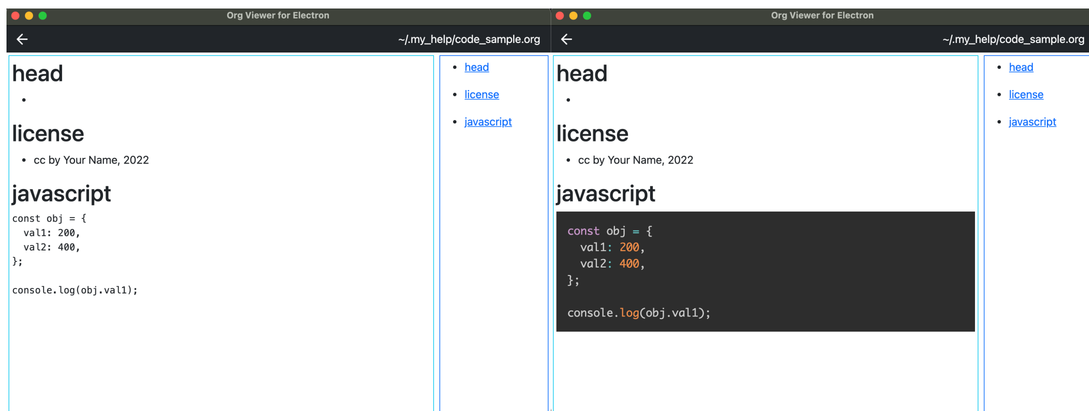

図 3.4: Prismjs を使用しない場合 (左) と，使用した場合 (右)

第4章　結論

今回の研究では，my_help の GUI 版となる org_viewer を開発した．また，このプロジェクトの github のリポジトリを公開し，実際にアプリケーションをインストールすることができるようにした．github にソースコードと簡単なマニュアルを公開した [11].

これにより，初学者が CLI に慣れていなくても，my_help の特徴を利用した効率的な学習が可能となる．

4.1　今後の課題

4.1.1　学習効率の計測

本研究では，実際に初学者に使ってもらい，学習効率が上昇するかどうかを確認することができなかった．そのため，初学者に実際に使用してもらい，必要な機能や改善が必要な機能を確認し，改良していくことができると考える．

4.1.2　テストの作成

本アプリケーションは，開発したプログラムが意図した動作をするかを手動で確認していた．しかし，ソースコードが多くなるにつれ，意図した動作をするかのテストを手動でするのは大変になる．そのため，テストコードを作成し，ソースコードを追加しても動作に影響がないかを確認する必要がある．javascript には jest[12] というテスティングフレームワークが存在するため，それを利用し，テストコードを作成することができる．また、オートメーションツールを利用したテストコードには、WebdriverIO もしくは Selenium を利用することができる [13]。これにより、GUI が適切に動作しているかをテストするコードを記述することができる。

謝辞

本研究を進めるにあたり，様々なご指摘を頂いた西谷滋人教授に深く感謝いたします．また本研究の進行に伴い，様々な助力や協力を頂きました西谷研究室の同輩，先輩方に心から感謝いたします．本当にありがとうございました．

参考文献

[1] daddygongon/my_help: Like CUI(CLI) help, the user makes and displays his own help, https://github.com/daddygongon/my_help , (accessed on 1 Jan. 2022).

[2] amake/orgro: An Org Mode file viewer for iOS and Android - github, https://github.com/amake/orgro , (accessed on 1 Jan. 2022).

[3] org-web, https://org-web.org/ , (accessed on 1 Jan. 2022).

[4] Electron — JavaScript, HTML, CSS によるクロスプラットフォームなデスクトップアプリ開発, https://www.electronjs.org/ , (accessed on 1 Jan. 2022).

[5] 【electron】electron+typescript+react でデスクトップアプリを作ってみる, https://qiita.com/yuu_1st/items/71c603c5958f97295bcc , (accessed on 1 Jan. 2022).

[6] React – ユーザインターフェース構築のための JavaScript ライブラリ, https://ja.reactjs.org/ , (accessed on 1 Jan. 2022).

[7] Bootstrap · 世界で最も人気のあるフロントエンドフレームワーク, https://getbootstrap.jp/ , (accessed on 1 Jan. 2022).

[8] Pandoc - About pandoc, https://pandoc.org , (accessed on 1 Jan. 2022).

[9] Prism, https://prismjs.com/index.html#features-full , (accessed on 1 Jan. 2022).

[10] averms/pandoc-filters: A small, useful collection of pandoc filters, https://github.com/averms/pandoc-filters , (accessed on 1 Jan. 2022).

[11] yuu-1st/org_viewer_for_electron, https://github.com/yuu-1st/org_viewer_for_electron , (accessed on 1 Jan. 2022).

[12] Jest · Delightful JavaScript Testing, https://jestjs.io/ja/ , (accessed on 1 Jan. 2022).

[13] 自動テスト - Electron, https://www.electronjs.org/ja/docs/latest/tutorial/automated-testing , (accessed on 1 Jan. 2022).

付 録A　webpackのソースコード

以下のコードは，https://zenn.dev/sprout2000/articles/5d7b350c2e85bc を参考に記述されている．

```
import path from 'path';

/** エディタで補完を効かせるためにに型定義をインポート */
import { Configuration, DefinePlugin } from 'webpack';

import HtmlWebpackPlugin from 'html-webpack-plugin';
import MiniCssExtractPlugin from 'mini-css-extract-plugin';

const isDev = process.env.NODE_ENV === 'development';

/** 共通設定 */
const base: Configuration = {
  mode: isDev ? 'development' : 'production',
  // メインプロセスで __dirname でパスを取得できるようにする
  node: {
    __dirname: false,
    __filename: false,
  },
  resolve: {
    extensions: ['.js', '.ts', '.jsx', '.tsx', '.json'],
  },
  output: {
```

```
    // バンドルファイルの出力先（ここではプロジェクト直下の 'dist' ディレクトリ）
    path: path.resolve(__dirname, 'dist'),
    // webpack@5.x + electron では必須の設定
    publicPath: './',
    filename: '[name].js',
    // 画像などのアセットは 'images' フォルダへ配置する
    assetModuleFilename: 'images/[name][ext]',
  },
  module: {
    rules: [
      {
/**
* 拡張子 '.ts' または '.tsx' （正規表現）のファイルを 'ts-loader' で処理
* node_modules ディレクトリは除外する
*/
test: /\.tsx?$/,
exclude: /node_modules/,
use: 'ts-loader',
      },
      {
/** 拡張子 '.css' （正規表現）のファイル */
test: /\.css$/,
/** use 配列に指定したローダーは *最後尾から* 順に適用される */
use: [
  /* セキュリティ対策のため（後述）style-loader は使用しない */
  MiniCssExtractPlugin.loader,
  {
    loader: 'css-loader',
```

```
    options: { sourceMap: isDev },
  },
],
      },
      {
/** 画像やフォントなどのアセット類 */
test: /\.(bmp|ico|gif|jpe?g|png|svg|ttf|eot|woff?2?)$/,
/** アセット類も同様に asset/inline は使用しない */
/** なお, webpack@5.x では file-loader or url-loader は不要になった */
type: 'asset/resource',
      },
    ],
  },
  /**
  * development モードではソースマップを付ける
  *
  * レンダラープロセスでは development モード時に
  * ソースマップがないと electron のデベロッパーコンソールに
  * 'Uncaught EvalError' が表示されてしまうことに注意
  */
  devtool: isDev ? 'inline-source-map' : false,
  plugins : [
    new DefinePlugin({
      'process.env.VERSION_ENV': `"${require('./package.json').version}"`,
    })
  ]
};

// メインプロセス用の設定
```

```
const main: Configuration = {
  // 共通設定の読み込み
  ...base,
  target: 'electron-main',
  entry: {
    main: './src/main.ts',
  },
};

// プリロード・スクリプト用の設定
const preload: Configuration = {
  ...base,
  target: 'electron-preload',
  entry: {
    preload: './src/preload.ts',
  },
};

// レンダラープロセス用の設定
const renderer: Configuration = {
  ...base,
  // セキュリティ対策として 'electron-renderer' ターゲットは使用しない
  target: 'web',
  entry: {
    renderer: './src/renderer.tsx',
  },
  plugins: [
    /**
     * バンドルした JS ファイルを <script></script> タグとして差し込んだ
```

```
    * HTML ファイルを出力するプラグイン
    */
    new HtmlWebpackPlugin({
      template: './src/index.html',
      minify: !isDev,
      inject: 'body',
      filename: 'index.html',
      scriptLoading: 'blocking',
    }),
    new MiniCssExtractPlugin(),
  ],
};

/**
* メイン，プリロード，レンダラーそれぞれの設定を
* 配列に入れてエクスポート
*/
export default [main, preload, renderer];
```

付 録B　hタグを抽出するソースコード

以下に，dom 要素から h1 から h3 タグを取得する関数を示す．なお，ソースコードは https://www.marorika.com/entry/create-toc を参考に，目的を満たすように改良したものである．

```
function HTMLCreateTableOfContents(dom: Document): HTMLDivElement {
  const result = document.createElement('div'); // 作成する目次のコンテナ要素
  // .h1, h2, h3 要素を全て取得する
  const matches = dom.querySelectorAll('h1, h2, h3');

  // 取得した見出しタグ要素の数だけ以下の操作を繰り返す
  matches.forEach(function (value, i) {
    // 見出しタグ要素の id を取得し空の場合は内容を id にする
    let id = value.id;
    if (id === '') {
      id = String(Math.random());
      value.id = id;
    }

    // 要素が h1 タグの場合
    if (value.tagName === 'H1') {
      let ul = document.createElement('ul');
      let li = document.createElement('li');
      let a = document.createElement('a');
```

```
      // 追加する<ul><li><a>タイトル</a></li></ul>を準備する
      a.textContent = value.textContent ?? '';
      a.href = '#' + value.id;
      li.appendChild(a);
      ul.appendChild(li);

      // コンテナ要素である<div>の中に要素を追加する
      result.appendChild(ul);
    }

    // 要素が h2 タグの場合
    if (value.tagName === 'H2') {
      let ul = document.createElement('ul');
      let li = document.createElement('li');
      let a = document.createElement('a');

      // コンテナ要素である<div>の中から最後の<li>を取得する.
      let lastUl = result.lastElementChild;
      let lastLi: Element | null;
      if (!lastUl) {
let ul2 = document.createElement('ul');
let li2 = document.createElement('li');
ul2.appendChild(li2);
result.appendChild(ul2);
lastUl = ul;
      }
      lastLi = lastUl.lastElementChild;

      // 追加する<ul><li><a>タイトル</a></li></ul>を準備する
```

```
      a.textContent = value.textContent ?? '';
      a.href = '#' + value.id;
      li.appendChild(a);
      ul.appendChild(li);

      // 最後の<li>の中に要素を追加する
      lastLi?.appendChild(ul);
    }

    // 要素が h3 タグの場合
    if (value.tagName === 'H3') {
      let ul = document.createElement('ul');
      let li = document.createElement('li');
      let a = document.createElement('a');

      // コンテナ要素である<div>の中から最後の<li>を取得する.
      let lastUl = result.lastElementChild;
      let lastLi: Element | null;
      if (!lastUl) {
let ul2 = document.createElement('ul');
let li2 = document.createElement('li');
ul2.appendChild(li2);
result.appendChild(ul2);
lastUl = ul2;
      }
      lastLi = lastUl.lastElementChild;
      let last2Ul = lastLi?.lastElementChild;
      if (!last2Ul) {
let ul3 = document.createElement('ul');
```

```
let li3 = document.createElement('li');
ul3.appendChild(li3);
lastLi?.appendChild(ul3);
last2Ul = ul3;
      }
      let last2Li = last2Ul?.lastElementChild;

      // 追加する<ul><li><a>タイトル</a></li></ul>を準備する
      a.textContent = value.textContent ?? '';
      a.href = '#' + value.id;
      li.appendChild(a);
      ul.appendChild(li);

      // 最後の<li>の中に要素を追加する
      last2Li?.appendChild(ul);
    }
  });

  return result;
}
```