

TypeScript でプログラマブルに動く日本語組版処理システムの提案

いなにわうどん @kyoto_inaniwa

<https://zenn.dev/inaniwaudon/articles/5d040f543c4c69>

はじめに

X（旧 X（旧 Twitter）のタイムラインが組版の話で盛り上がっていたため、自分も軽く参画したところ思いのほかアツくなってしまい、今なお組版への熱が失われていなかったことを再確認した 12 月初頭です。

さて、春先に TypeScript 上にてプログラマブルに作動する日本語組版処理システム（以下、仮称として minitype を用います）を構想し、数週間掛けてプロトタイプの実装を行ってきました。ところが、今年度になって個人開発にリソースを割く余裕がなくなり、宙ぶらりんな状態のまま年末を迎えてしました。まだ開発途中ではありますが^{*1}、折角なので「日本語組版処理システムの夢^{*2}」としてアイデアを供養するとともに、具体的なプロトタイプの実装を示したいと思います^{*3}。

実装についてはソースコードを見ていただくこととして、本記事では特に「ユーザがどのような記述をするとどのような出力が得られるのか」という点に焦点を絞って紹介を進めていきます。

概要

minitype は TypeScript を用いて記述され、Node.js 上で作動します。プロトタイプの実装を以下の GitHub レポジトリに公開しています。開発途中であるため機能は限定的です（バグもあります）。

- [inaniwaudon/minitype-test](https://github.com/inaniwaudon/minitype-test)
<https://github.com/inaniwaudon/minitype-test>

詳細を話す前に、まずはシステムの動作例をご覧ください。以下のソースコードは、本記事と同様の文書を記述した ts ファイルになります。minitype のパッケージを `npm i` した後、

^{*1} ソフトウェアは出す時期も大事で、完璧を目指すと一生日の目を見ることはない

^{*2} ヨドバシの福袋？

^{*3} 概念実証のような段階であり、実用は目的としていません

`npx article.ts` を実行することにより、図 1 に示す PDF 文書を出力することができます。

(ソースコードは省略)

The image shows two side-by-side code editor panes. The left pane contains TypeScript code for defining document structures like Paragraph and HeadingLevel. The right pane shows the resulting PDF document with Japanese text and styling.

```

TypeScript でプログラマブルに動く日本語組版処理システム

TypeScript でプログラマブルに動く日本語組版処理システムの提案

はじめに
X (目次 Twitters) のタイムラインが組版の面倒が上がっていたため、自分も経験したところ思いのほどのアツくなってしまい、今なお組版への熱が失われていなかったことを再確認した 12 月前頃です。
さて、春先に TypeScript 上にてプログラマブルに作動する日本語組版処理システム（以下、仮称として minitype を用います）を構想し、数週間かけてプロトタイプの実装を行っていました。ところが、今年度になって個人開発にリソースを割く余裕がないなり、留められない状態のまま半年を迎えてしましました。まだ開発途中ではありますか*1、折角なので「日本語組版処理システムの夢*2」としてアイデアを供養するとともに、具体的なプロトタイプの実装を示したいと思います*3。
実装についてはソースコードを見ていただくこととして、本記事では特に「ユーザなどのような記述をするとどのような出力が得られるのか」という点に焦点を絞って紹介を進めていきます。
概要
minitype は TypeScript を用いて記述され、Node.js 上で作動します。プロトタイプの実装を以下で GitHub レポジトリに公開しています。開発途中であるため機能は限定的です（バグもあります）。
inaniwaudon/minitype-test  

https://github.com/inaniwaudon/minitype-test
詳細を語る前に、まずはシステムの動作例をご覧ください。以下のソースコードは、本記事と同様の操作で実行できます。
*1 ソフトウェアは出力時も大事で、完璧を目指す一日目の見ることはない
*2 モドックの構築
*3 概念実証のような段階であり、実用は目的としていません

```

```

TypeScript でプログラマブルに動く日本語組版処理システム

同様の文書を記述した ts ファイルになります。minitype のパッケージを npm i した後、npx article.ts を実行することにより、以下の PDF 文書を出力することができます。（ソースコードは省略）

実装としては、OpenType の読み込みに opentype.js を、PDF の描画に pdfkit を使用しています。現状の実装ではテキストをアウトライン化した PDF を生成しているため、フォントの埋め込みは今後の課題になります。加えて、シンタックスハイライトに lowlight を、フォントのキャッシュ周りには sqlite3 を使用しています。

マークアップ
minitype は独自のマークアップ言語を持ちません。その代わりに、TypeScript (JavaScript) のオブジェクトを用いてマークアップが行われます*4。以下に、本文および見出しを表すブロック（ページを縦方向に構成する要素）の型定義を示します。ユーザはブロックを記述する際に、この型に従ったオブジェクトを作成します。なお、ブロックには他にも（現時点にて）リスト、図、キャプション、脚注等が存在します。

```

```

export type Paragraph = {
  type: "paragraph";
  lines: Inline[][];
  style?: Partial<textStyle>;
};

type HeadingLevel = 1 | 2 | 3 | 4;
export type Heading = {
  type: "heading";
  level: HeadingLevel;
  lines: Inline[];
  style?: Partial<textStyle>;
};

最终的な組版を行う際、ユーザは以下の minitype 関数を呼び出します。第一引数にはブロックから構成される配列を渡します。見出しと本文からなる簡単な文書を作成するには、以下の通りに記述を行います。

```

```

const body = [
  ...
];
*4 雜に表現するとすべては JSON で表現できるということです

```

図 1 : minitype を用いて作成した組版例

実装としては、OpenType の読み込みに `opentype.js` を、PDF の描画に `pdfkit` を使用しています。現状の実装ではテキストをアウトライン化した PDF を生成しているため、フォントの埋め込みは今後の課題になります。加えて、シンタックスハイライトに `lowlight` を、フォントのキャッシュ周りには `sqlite3` を使用しています。

マークアップ

`minitype` は独自のマークアップ言語を持ちません。その代わりに、TypeScript (JavaScript) のオブジェクトを用いてマークアップが行われます*4。以下に、本文および見出しを表すブロック（ページを縦方向に構成する要素）の型定義を示します。ユーザはブロックを記述する際に、この型に従ったオブジェクトを作成します。なお、ブロックには他にも（現時点にて）リスト、図、キャプション、脚注等が存在します。

```

export type Paragraph = {
  type: "paragraph";
  lines: Inline[][];
}

```

*4 雜に表現するとすべては JSON で表現できるということです

```
    style?: Partial<TextStyle>;
};

type HeadingLevel = 1 | 2 | 3 | 4;
export type Heading = {
    type: "heading";
    level: HeadingLevel;
    lines: Inline[][][];
    style?: Partial<TextStyle>;
};
```

最終的な組版を行う際、ユーザは以下の `minitype` 関数を呼び出します。第一引数にはブロックから構成される配列を渡します。見出しと本文からなる簡単な文書を作成するには、以下の通りにスクリプトを記述します。

```
import { minitype } from "minitype";
const body = [
    {
        type: "heading",
        level: 1,
        lines: [["見出し"]]
    }, {
        type: "paragraph",
        lines: [["本文だよ～～ 2024 年が終わります！！"]]
    }
];
const documentStyle = { ... } // 省略
minitype(body, documentStyle, "output.pdf");
```

このスクリプトを `tsx` を用いて実行することにより、1-2 秒程度で PDF 文書である `output.pdf` が得られます。

なぜ TypeScript?

今日の多くの組版処理システムでは、組版処理エンジンとマークアップ言語は不可分の関係にあります。例えば `LaTeX` を語る際に、組版処理エンジンとかの `TeX` 表記を分離することは困難です。一方、言語の自作から着手すると、組版の要件に対する理解のほかに言語処理系への知見が求められ、開発コストが増大してしまいます^{*5}。また、ユーザは組版処理システム

^{*5} パーサを書く必要があるのは勿論のこと、例えば LSP (Language Server Protocol) どうするの？みたいな話が出てくる

の数だけ言語を覚える必要があり（あるいは、その言語に合わせたトランスパイラを書く必要があり）厄介です。

そこで、minitype は入力として JavaScript 標準のオブジェクトを採用することにより、これらの短所を克服します。ユーザが文書を記述する際は静的型付けによる支援を受けられるほか、現在であれば Copilot 等のツールも活用できるはずです。

記述のカスタム

オブジェクトを逐一記述するのが面倒に感じた場合には、適当なヘルパ関数を定義すればよいです。以下のソースコードに、タイトルや見出しを記述するためのヘルパ関数を示します。これにより、`title()`, `headline()` 等の記述で簡潔に見出しを作成することができます。ヘルパ関数は、文書の用途に合わせて柔軟に定義することができます。また、Unified 等のエコシステムを用いて Markdown や独自言語からのトランスパイルすることも可能です。

```
import { Heading, HeadingLevel, Paragraph, TextAlign } from
"minitype";

const heading =
  (level: HeadingLevel) =>
  (text: string): Heading => {
    return {
      type: "heading",
      level: level,
      lines: [[text]],
    };
  };
const title = heading(1);
const headline = heading(2);

title(" タイトル ");
headline(" 見出し ");
```

プログラマブルな文書作成

minitype では、文書構造を TypeScript のソースコード中に直接記述するため、プログラマブルな処理が得意です。以下に、zipcloud の郵便番号 API を利用して、郵便番号と住所の対応を出力する例を示します。

```
const addresses = [];
for (const postcode of ["3050045", "3050005", "3050821",
"3050006", "3002648"]) {
    const response = await
fetch(`https://zipcloud.ibsnet.co.jp/api/search?zipcode=$
{postcode}`);
    const json = await response.json();
    const result = json.results[0];
    const address = result.address1 + result.address2 +
result.address3;
    addresses.push([postcode, address]);
}
const body = addresses.map(([postcode, address]) =>
    p(` ${postcode}: ${address}`)
);
```

3050045 : 茨城県つくば市梅園

3050005 : 茨城県つくば市天久保

3050821 : 茨城県つくば市春日

3050006 : 茨城県つくば市天王台

3002648 : 茨城県つくば市豊里の杜

図 2 : 郵便番号と対応する住所の出力例

このように、前処理を行うことなく、動的に反復処理を行ったり、Fetch API を利用して Web API から取得したデータを処理したりすることができます。感覚としては React に近いですね。また、関数呼び出しによって組版処理を実行できるため、Web サーバ等からシームレスに実行されることも期待されます。

スタイリング

テキスト等のブロックには、フォント等のスタイルを指定することができます。以下に、スタイルの型定義以下を示します（一部未実装のスタイルもあります）。面白いプロパティとしては、文字揃えを指定する align プロパティが挙げられます。これはページ番号を引数に取るラムダ式を取ることが可能であるため、`(pageIndex % 2 === 1 ? "left" :

"right")` のように戻り値を指定することにより、奇数ページは左揃え、偶数ページは右揃え、といった表現が可能となります。その他に、段落を矩形で囲う等の装飾も実現できます。

```
export type TextStyle = {
    lineHeight: number;
    align?: TextAlign | ((pageIndex: number) => TextAlign);
    indent?: number | Em;
    firstIndent?: number | Em;
    pre?: boolean;
    space?: TextSpace;
} & InlineStyle;

export type InlineStyle = CharStyle & BoxStyle;
export type CharStyle = {
    size: number;
    font: string;
    color: Color;
    baseline?: number;
};
export type BoxStyle = {
    margin?: Margin;
    padding?: Padding;
} & Decoration;
export interface Decoration {
    background?: Color;
    border?: AllBorder;
    borderRadius?: number;
}
```

minitype はプログラマブルな指定が特徴的だと述べましたが、これはページの版面指定にも現れています。例えば余白に最低 30 mm 確保しつつ、本文サイズの等倍になるように本文幅を指定したい場合、以下のソースコードによって実現されます。

```
const paragraphSize = 3.5;
const padding = (() => {
```

```

const bodyWidth = Math.floor((210 - 30 * 2) / paragraphSize)
* paragraphSize;
const horizontal = (width - bodyWidth) / 2;
return {
  top: 32,
  right: horizontal,
  bottom: 30,
  left: horizontal,
};
})();

```

なお、ブロックへのスタイルは一括指定するほかに、個々のブロックに対して個別指定することも可能です。また、ブロック間の間隔は、以下の通りに指定されます。この場合 h1-paragraph ブロック間には 2 mm、paragraph-h2 間には 8 mm の余白が生じます。fallback は全てのブロックを表すため、h2- 全ブロックの間には 2 mm の余白が生じます。

```
[
  ["h1", "paragraph", 2],
  ["h2", "fallback", 2],
  ["paragraph", "h2", 8],
]
```

コマンド

文字単体に装飾を加えたい場合は、コマンドを使用することができます。paragraph や headline はブロックは `lines` プロパティにインライン（文字列またはコマンド）からなる 2 次元配列のプロパティを取ります。例えば、一部分を太字に設定したい場合は、以下のように記述をします。

```

[
  ["ここは細い字なんだけど", { name: "b", body: ["ここは太字"] }, "で
表現される"]
]

// スタイル定義部
{
  b: { font: "AP-SK-IshiiGothicStdN-EB" },

```

}

日本語組版

日本語組版に関しては、基本的なベタ組み（追い出し）のアルゴリズムを実装しています。文字アキ量設定に関しては、簡易的な実装ではあるものの、ユーザによる調整が可能です。ルビ、割注、カーニング、異体字といった複雑な組版機能は今後対応予定です。

むすびに

年末ということで、TypeScript 製の組版処理システムを紹介してみました。加えて、minitype を用いれば、Zenn に書く程度の内容の文書であれば組版が可能であることを示しました。記事では不完全な説明に留まっているため、ご興味があればソースコードを覗いていただければと思います。本システムの利用例としては、簡単なレポートの執筆や、GitHub Actions 等の CI を活用したブログ記事の自動組版等を想定しています。

とはいっても、縦組みも二段組も表組みもなく、LaTeX や SATySFi、Typst といった先発の組版処理システムには足元にも及ばない状態です。卒論もあり、しばらくは多忙な日々が続きそうなのですが、今後もスローペースで開発を継続していければと思います。2024 年も残り僅か、どうぞ良いお年をお過ごしください。