日本ソフトウェア科学会第 40 回大会 (2023 年度) 講演論文集

Minissg: 小さく軽量で規約のない静的 Web サイトジェネレータ

上野 雄大

本論文では，従来の Web 関連技術がプログラミング言語の外に課す規約を徹底的に排除した静的サイトジェネレータ Minissg を提案する．Minissg の中心的な設計思想は，最新の汎用言語とその汎用ツール群こそが，現時点における最高の開発体験と検証体系を与える最大の源泉である，というものである．Minissg は TypeScript で書かれた約 1,100 行の小さな Vite プラグインであり，言語それ自体が本来持つ汎用の抽象・部品化機構や整合性検査機能を，Web サイト生成の全局面でそのまま利用可能にする．これにより，ユーザーに何も提案しない代わりにユーザーのあらゆる選択を尊重するという，Web 以外では至極当然な開発体験を Web サイト生成に導入する．本稿では Minissg の設計と実装上の判断を述べ，他のフレームワークと比較する．また，Astro で構築した約 8,700 行の Web サイトを Minissg を用いてほぼ同規模で再構築した事例を紹介し，その有用性を確認する．

This paper presents “Minissg,” a static website generator that does not impose any choice and convention for coding. Its policy different from others is that it aims to write down everything inside a website only with the unmodified features of a state-of-the-art general-purpose programming language and its tools. Minissg is a small Vite plugin of about 1,100 lines of code in TypeScript. This small code base provides the users with nothing but the maximum freedom of design choices in exchange of a little effort of writing everything in a code. This paper describes the choices on the design and implementation of Minissg, including comparison with other frameworks. This paper also discusses its usefulness through an example of translating a website of about 8,700 lines of code with Astro into one of almost the same size with Minissg.

1 はじめに

統一的な見た目と使用感を持つ Web サイト制作の効率化や CDN（Contents Delivery Network）による高速配信の実現などのために，静的な Web サイトをプログラムで自動生成することが近年広く行われている．ここでいう「静的」とは，Web サーバがクライアント（Web ブラウザ）からの要求を受け取る前に，要求に対する応答の内容があらかじめ定まっていることを言う．静的な Web サイトの公開は，事前に生成した HTML ファイルを Web サーバのファイルシステムに配置することによってのみ行われる．Web サーバは Web ブラウザにファイルを返すだけでよく，効率面ではスループットの向上が期待でき，安全面でもセキュリティリスクの低減が見込める．静的な Web サイトの自動生成することを static site generation，そのためのソフトウェアを static site generator といい，ともに「SSG」と略される．

SSG の中核的な役割のひとつは，統一的な見た目と使用感を簡潔な記述で実現するために，Web ページ間で共通するレイアウトやメニューなどを共有する機能を提供することである．例えば Jekyll [8] は，ディレクトリ内に含まれる Markdown ファイルそれぞれについて，その内容をいくつかのメタデータや共通部品と共に Liquid [14] で書かれたテンプレートに埋め込むことを，その主要な機能としている．ユーザは，Markdown ファイルごとに埋め込み先のテンプレートファイルを指定する機能や，Liquid の `include` タ

Minissg: a Small and Lightweight Static Site Generator without Convention.

Katsuhiro Ueno, 新潟大学自然科学系, Academic Assembly Institute of Science and Technology, Niigata University.

グなどによる簡易的なモジュール機構を用いて，サイトメニューなどの Web サイト共通の構造を Web ページ間で共有する．

これらの共有機能は，現代的な汎用のプログラミング言語ならばどの言語でも有しているような，関数やモジュールなどの抽象機構に他ならない．この点を踏まえてプログラミング言語の視点から SSG を見ると，これまでの SSG や，SSG に含まれるテンプレート言語は，コンテンツを書くことと，書いたコンテンツをプログラミングの技法を用いて管理・操作することを両立しようとした記述体系である，と見ることができる．記述体系としてのこれらのシステムの特徴は，ユーザがコンテンツを書くことに集中できるように，ユーザが書くコードの量を最小限にする工夫が見られることである．そのひとつが，“convention over configuration” と呼ばれる，記述体系の外側に存在する構造が記述対象の意味の一部を定める方式である．本論文では，このような記述システムの外に関する約束事を「規約」と呼ぶ．例えば，ソースコードのファイル名から処理の対象や内容を暗黙に決定したり，特定の形式に従う変数名にファイルやデータベースの内容を暗黙に対応付けるのは，規約の例である．

規約は，確かにコードの記述量を削減するものの，その一方で，それが十分な汎用性，堅牢さ，および生産性を併せ持つプログラミングの枠組みを実現しているかには，疑問の余地が残る．記述言語の外側にある規約と，記述言語の内側にある制御・抽象機構の関連は，明確とは言いがたい．プログラムを読解するにはフレームワークごとに微妙に異なる暗黙の規約を理解しておかねばならず，言語の意味論に従ってコードを読むだけではプログラムを読み解くことができない．そのため，コード全体の整合性を静的に検査することも，理論と実践の両面において，困難である．

もしプログラミング言語が十分な機能を有しているならば，一般のプログラム開発と同様に，Web サイトの制作も言語に閉じた記述によって達成し得るはずである．SSG の文脈においては，この十分な機能とは以下の 3 点にまとめられる．

1. Web ページの内容の製作に集中できるコンテンツ記述システム
2. 汎用プログラミング言語によるコンテンツの生成および合成機構
3. Web サイトを構成するファイル間の関連付けとその管理方式

このうち 1 と 2 については，コンテンツ自体をストレスなく記述できる言語と，記述されたコンテンツを第一級市民として扱える言語機構が必要である．これらは必ずしも両立することではない．しかし，幸いなことに，今日では，近年の JavaScript の猛烈な発展により，JavaScript とその周辺環境は，統合された Web 開発言語として，2 だけでなく 1 をも満たしつつある．例えば，JavaScript に HTML 類似の木構造構成マクロを加えた拡張言語である JSX [10] は，コンテンツと制御・抽象構造の両方に対する高い表現力を備えており，モジュールや高階関数を含む汎用のソフトウェア設計技法を Web コンテンツの構成に応用することに成功している．また，Markdown を JSX で拡張した MDX [5] は，Markdown に基づく標準的で簡潔な記法で書かれた文書に対して JSX の意味論を与え，データとしてもコードとしても解釈できるコンテンツの記述を実現している．残る 3 についても，JavaScript のビルドツールによってすでに取り組みが行われている．もはや，JavaScript とその周辺環境は，Web プログラミングに規約を持ち込む必要がないほどに成熟していると考えられる．そうであるならば，特別なテンプレートシステムを別途導入しなくても，JavaScript のみで，JavaScript の豊富なツールの支援を享受しながら，Web サイトを制作できるはずである．

以上の分析に基づき，本論文では，Web サイトの全てを生成するプログラムを JavaScript やその拡張言語のみを用いて言語の中で書き切るためのシステム Minissg[†1] を提案する．Minissg は，JavaScript とその周辺ツールの全ての機能を，何の改変や隠蔽を行う

†1 著者はこの名前を英単語の “missing” と同じ発音で呼んでいる．当初は “Mini SSG” という仮の名前をつけていたが，開発中に “missing” と頻繁に混同してしまったことが面白く，また本開発は Web 制作の missing piece を埋めるものであるという気持ちとも掛けて，仮名をそのまま正式な名前として，敢えて “missing” と見間違えやすい表記で採用した．

ことなく，そのまま SSG に導入する．従って Minissg はユーザに何の規約も課さず，何のライブラリやツールの導入も強制しない．一般のプログラム開発と同様，プログラムファイルの配置やライブラリの選定，モジュール設計に至るまで，Web 制作に関わるあらゆる決定はプログラミング上の決定としてユーザに委ねられる．コードフォーマッタやリンタを含む開発支援ツールについても，既存の汎用のものをそのまま用いることができ，それらの選択もユーザの自由である．

JavaScript の成熟により，Minissg の目標のほとんどは，すでに JavaScript とその周辺技術によって達成されている．Minissg が独自に行う必要がある仕事は，画像ファイルやスタイルシートなどの資源管理と，それらを参照する HTML ファイル群の生成に限られる．Minissg の目標は，これらを Minissg が暗黙に行うことではなく，ユーザが JavaScript のコードとして無理なく自在に書けることである．著者は，この目標の達成に向けて，既存のツールが提供する JavaScript プログラミング環境を分析し，Minissg が SSG システムとして提供する機能を厳選した．その結果，Minissg は，Web のフロントエンド開発で広く用いられているツールのひとつである Vite [18] に対して，SSG のためのわずかな機能を加えるプラグインとして，TypeScript で 1,100 行程度の分量で実装された．この小ささは，他の SSG に対する Minissg の優位点のひとつである．Minissg の実装を読み解いて理解したり，必要に応じて Vite 以外のツールに移植・移行したりすることは，他の SSG と比較してはるかに容易であろう．

著者は，この小さな Vite プラグインがフルスタックフレームワークに匹敵する生産性を持ち得ることを，Astro [1] で書いた Web サイトのひとつを Minissg を用いて書き直すことで確認した．書き直し元の Web サイトには，データベースからのページ生成，ページネーション，多言語対応など，SSG による一般の Web サイト構築に現れる典型的な状況が含まれる．結果として，わずか 2%のコードサイズの増加で，Minissg への書き換えを終えることができた．この結果から，少なくともこの事例に類する Web 制作においては，Minissg がフルスタックフレームワークと同等の生産性を有していると言える．

本論文では，Minissg の設計と実装，前述した書き換え事例の報告，および他の SSG との比較を含めた，Minissg の開発に関わる一連の経験を報告する．本論文の構成は以下のとおりである．2 節では，JavaScript とその周辺ツールを用いた現代的な Web サイト開発手法を概観したのち，Minissg で実現する SSG のための機能を洗い出す．3 節では，Minissg の設計・実装上の主要な着眼点を，Minissg の簡単な利用例を通じて紹介する．4 節では，実用性を高めるための追加機能について論じる．5 節では，Astro から Minissg への書き直し事例の概要を報告する．6 節では，Minissg を既存の SSG や Web フレームワークと比較し，その独自性と有用性を確認する．7 節は本論文のまとめである．

Minissg のソースコードや簡単な使用例は `https://github.com/uenoB/vite-plugin-minissg/` から取得できる．Minissg の詳しい使い方については，ソースコードに含まれる `README.md` をご参照願いたい．

2 JavaScript による Web サイト開発の方針

JavaScript は，Web ブラウザ上で動作するスクリプトを記述するためのプログラミング言語であったが，今日においてはそれに留まらず，サーバー側のプログラムや，Web とは無関係のコマンドを実装するための汎用のプログラミング言語としても用いられている．Web に由来する厚いユーザ層によって，JavaScript を用いたソフトウェア開発について様々なアイディアが提案・具現化され，その結果得られた一連のソフトウェア資産は，パッケージ管理システムの下で誰でもすぐに利用できる形で提供されている．

JavaScript を取り巻く環境の特徴のひとつは，JavaScript プログラムを解析・変換するためのメタなライブラリやツールが，JavaScript で書かれたプログラムとして，豊富に開発・提供されていることである．これらを利用することで，JavaScript の拡張言語の処理系や，JavaScript を核とする開発環境を，JavaScript 自身で実装することができる．今日の Web フロントエンドの開発は，これら JavaScript

で実装された JavaScript の拡張言語や開発環境を用いて行うのが一般的である．例えば，ソースコードを素の JavaScript で直接書くのではなく，型による静的検証を追加した言語 TypeScript や，種々の構文糖を展開する変換器 Babel が導入する拡張言語で書き，webpack や Vite などのビルドツールを用いて，スタイルシートや画像などのデータと共に，Web 上で実行可能な形式に変換・結合する．この例に現れるツールは全て JavaScript で書かれている．JavaScript でプラグインを書くことでツールを更に拡張することも可能である．

Minissg が目指すのは，これらの充実したツール群による恩恵の自然な延長線上に，静的な Web サイトを生成するプログラムを書くためのプラットフォームを実現することである．そのためには，Minissg を，すでに広く受け入れられている言語拡張と親和性の高い形で，JavaScript による JavaScript の拡張体制の一部として，設計・実装するべきであろう．著者は，Minissg を実現するための基盤として，すでに広く用いられているビルドツールのひとつである Vite [18] を選択した．その理由は以下の 3 点である．

1. 製品版のバッチビルドだけでなく，開発中の確認のためのオンデマンドなビルド機能（開発サーバ機能）を有しており，待ち時間の少ない開発サイクルを回すことができること．この特徴は，生成する Web サイトの規模にかかわらず迅速にプレビューを得ることができるという点において，SSG においても有用と期待できる．
2. プラグイン機構がよく整理されており拡張が容易なこと．Vite のプラグイン機構は，ソースファイルの探索・読み込み・変換・結合など，ビルドに関わる全ての工程への介入と拡張を許しており，Vite 自身もこの機構によるプラグインの集まりとして実装されている．また，プラグイン機構自体も他のビルドツールとの互換性を重視して設計されており，ドキュメントやリファレンス実装が充実している．
3. 豊富な利用実績があり，ユーザの広い指示を集めていること．JavaScript のツールは世代交代が速く，今の流行が必ずしも将来の安寧に繋がるとは限らない．従って，今後の幅広い利用を想定し，ソフトウェアとしての寿命をできるだけ長く得るためには，現時点で最も先進的なツールのうち，コミュニティが大きく継続的な開発と利用が期待できるものを選ぶのが最善である．Vite は，本稿執筆時点では名実共に，その最善の選択肢のひとつである．

```
import ViteLogo from ’./vite.svg’
import { logo } from ’./App.module.css’

export default function App() {
  return (
    <a href="https://vitejs.dev">
      <img className={logo}
           src={ViteLogo} alt="Vite"/>
    </a>
  );
}
```

図 1　データのインポートを含む JavaScript コード例

Vite は JavaScript のモジュール機能を，JavaScript 以外で書かれたソースコードや，そもそもプログラムではないデータにも拡張する．Vite はデータファイルの JavaScript コードとしてのインポートをファイル名のパターンマッチなどによって識別し，それらを「そのデータを生成しその効果を発現する」ことを副作用として行う JavaScript コードとして取り扱う．ユーザは，この拡張されたモジュールシステムを用いることで，コードそのものだけでなく，コードからデータへの参照も，JavaScript の意味論の下で管理することができる．

例えば，図 1 は画像への参照とスタイルシートの適用を含む HTML 断片を返す関数の実装例である．最初の 2 行には，他の JavaScript モジュールを利用することを表す `import` 文が書かれているが，それらのインポート先のファイルは SVG 形式の画像ファイルと Module CSS 形式のスタイルシートであり，共にプログラムですらない．Vite は `import` された画像ファイルを，「その画像ファイルを Web ブラウザから見える箇所に置く」という副作用をトップレベ

ルに持ち，その画像ファイルへのパスをエクスポートする JavaScript モジュールとして解釈する．また，スタイルシートの `import` も，「そのスタイルをこのコードの実行を司る Web ページにグローバルに適用する」という副作用を持つモジュールのインポートと解釈する．`App` 関数は JSX による構文糖を用いて書かれており，適切なプラグインが設定されていれば，Vite は自動的にこの構文糖を展開する．このコードをビルドした結果として，Web ブラウザが直接解釈できる形式の画像ファイル，CSS ファイル，および JavaScript ファイルが生成される．さらに，その CSS ファイルと JavaScript ファイルへの参照を，それぞれ `link` 要素および `script` 要素として，トップレベルの HTML ファイルに挿入する．Web ブラウザでこの HTML ファイルを読み込むと，データファイルのインポートに対する拡大解釈も含めて，あたかもプログラムが Web ブラウザ上で実行されているかのように見える．

この `import` 文の拡大解釈は，JavaScript の意味論の自然な拡張として，JavaScript によるコードの記述によってデータセットを管理することを実現している．コードとデータの関連がコードとして拡張言語の中で記述されているという点において，この拡張は，言語の外で意味を拡張する規約とは異なる．どのファイル名をデータのインポートと認識するかは Vite の設定に依存するものの，その設定も天下り的に固定で与えられる規約ではなく，ユーザが変更可能な設定項目のひとつである †2．これ以降，本論文で導入する様々な言語拡張も，言外の意味を含む規約ではなく，必ず言語の中で記述を伴う形で導入している．

この拡張は，SSG において，プログラムが生成する HTML ファイルに，その HTML ファイルから参照されるデータを関連付ける機構として，そのまま採用することができる．この拡大解釈は既存のビルドツールで達成済みであり，すでにユーザの指示を集めて広く利用されているという事実も，Minissg が掲げる「全てをコードで書き下す SSG」の実現にとって都合が良い．筆者は，以上の分析から，Minissg の典型的な利用シナリオを以下の 3 ステップと定めた．

1. ユーザは，Web サイトを構成する HTML ファイル群を生成するプログラムを，Vite が導入するモジュールシステムの拡張の下で，JavaScript やその拡張言語を用いて記述する．以下，このプログラムを「生成プログラム」と呼ぶ．生成プログラムは Vite によって解析・結合されるので，その記述には，Vite から利用できるあらゆる機能や拡張を活用することができる．Minissg 自体も Vite プラグインとして実装し，ユーザは Vite の設定ファイルで Minissg を有効にできるものとする．
2. Minissg は，生成プログラムを Vite でビルドした後，そのプログラムを実行し，データへの参照を含む HTML ファイル群を生成する．それぞれの HTML ファイルに含まれるデータの集合は，その HTML ファイルを生成する過程で実行された「データファイルのインポート」による副作用を通じて取得・管理する．モジュールのインポートによる副作用は，JavaScript の意味論においてはそのモジュールを初めてインポートした時にしか発生しないため，生成プログラムを複数の HTML ファイルを一括で生成するプログラムとして一度だけ実行してしまうと，それぞれの HTML ファイルとそれ固有のデータとの関連が不明瞭になってしまう．これを解消するため，生成プログラムの実行は，概念上，生成対象の HTML ファイルごとに個別に実行するものとみなす．ある特定の HTML ファイルを生成する実行パスの上でインポートされる一連のデータのみを，その HTML ファイルと関連付けられたデータとする．
3. 生成された HTML ファイル群を入力として Vite を再度実行する．Vite の標準機能によって，一連のファイルが Web ブラウザから直接アクセスするのに適した形式に変換・整形され，静的な Web サイトが完成する．

このシナリオの大部分が，すでに Vite によって達成されていることの再利用であることに注意されたい．これは，Minissg が JavaScript やその周辺ツール

†2 設定ファイルの記述言語も JavaScript である．

の機能をそのまま SSG に持ち込むことの証左であり，また Minissg がごく小さなプラグインとして SSG を達成できた理由である．結局のところ，このシナリオを実現するために Minissg が独自に設計・実装しなければならないことは，以下の 3 点に限られる．

1. 生成される HTML ファイル群を JavaScript で表現するためのデータ形式
2. 「生成プログラムを HTML ファイルごとに個別に実行する」ことの効率的な実現
3. 生成プログラムの実行と 2 度の Vite の起動を含むビルドプロセス全体の構成

3 Minissg の設計・実装上の要点

本節では，前節で得た Minissg の設計・実装項目に対して筆者が行った判断を，項目ごとに小節に分け，Minissg の利用例を交えながら報告する．

3.1 生成ファイル群の表現方法

ファイル配置の管理はユーザの使用感に直結する仕様であり，だからこそ多くの SSG が規約によって実現している項目である．Minissg の設計においては，JavaScript の本来の意味論や記述能力に注意するだけでなく，典型的な利用局面の想定を含めた，総合的な判断が必要である．

Web サイトにおいて HTML ファイルは，Web ブラウザによってそれぞれ個別に解釈され，独立したユーザインターフェースを構成する．従って，ユーザは，典型的には，各 HTML ファイルを生成するプログラムを互いに独立したモジュールとして書きたいはずである．また，各 HTML ファイルの Web サイト上でのパスを，そのファイルを生成するモジュールの名前や階層に合わせるのは，ごく自然な発想であろう．既存の SSG や Web フレームワークの多くは，この自然な発想をファイルベースのルーティングと呼び，規約のひとつとしてユーザに課している．

Minissg が目指すのは，この自然な発想を，省略する必要がない程に直截で短いコードとして書ける記述体系である．これを達成するために，Minissg は以下の 2 つの機能を利用する．

1. JavaScript のモジュールが一般のオブジェクトと同型であるという仕様
2. Vite が導入した `import.meta.glob` 機能によるワイルドカードを用いた動的インポート機構 1 を活用することで，モジュールの木構造をオブジェクトの入れ子構造として表現・操作することが可能である．この木構造を，生成される HTML ファイルのパスと内容を対応付けるデータ構造として用いる．さらに 2 を利用し，ワイルドカードによるファイル選択と正規表現による文字列操作で，モジュールの一括ロードとパスの決定を直截なコードで実現する．

```
const mdFiles = import.meta.glob(
  "./pages/**/*.md",
  { query: { render: "" } }
);
export const get = () =>
  new Map(
    Object.entries(mdFiles)
      .map(([name, importFn]) =>
        [name
          .replace(/^.*\//, "")
          .replace(/\.md$/, ".html"),
         { get: importFn }])
  );
```

図 2 Minissg で `pages` ディレクトリ以下の `.md` ファイル群から HTML ファイル群を生成する例

例を通じて本項目に関する Minissg の設計判断を説明する．典型的な利用局面として，プロジェクトの `pages` ディレクトリ以下を再帰的に辿って到達できる全ての `.md` ファイルを，同じ名前で拡張子だけ `.html` に置き換えた HTML ファイルに変換することを想定する．Minissg ではこのことを，Vite に `.md` ファイルを JavaScript コードとしてインポートするプラグインを設定した上で，図 2 に示すコードを書くことで実装できる．Vite が提供する `import.meta.glob` 関数は，指定されたワイルドカードにマッチする全てのファイルそれぞれについて，そのファイル名からそのファイルを動的にインポートする関数へのマップをオブジェクトとして返す．第 2 引数はコンポーネントを文字列に変換する Minissg の拡張機能を適用するた

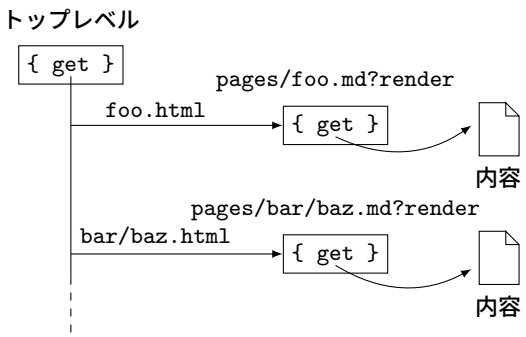

図 3　モジュールの木構造で表現された生成 HTML ファイル群

めの設定であり，詳しくは 4.1 節で説明する．get 関数は，このモジュールを根とする部分木の生成の遅延を意味する関数であり，Minissg が木構造を辿るときに必要に応じて呼び出される．get 関数内では，正規表現による文字列置換によって，import.meta.glob が返したマップを，HTML ファイル名からその内容を生成する関数へのマップに変形する．このソースファイル全体は，マップを返す get 関数をエクスポートするモジュールであり，それは get メソッドだけを持つオブジェクトと同型である．結果として，このソースファイルは，図 3 に示すようなモジュールの木構造を表す．ただし，各部分木の生成は get 関数によって遅延されている．この木構造は，全ての部分木が生成された暁には，生成される HTML ファイルのパスからその内容へのマップを表現する．葉が HTML ファイルの内容を表し，根からその葉に至るパスに付けられたラベルがそのファイル名を表す．

この方式は，この典型例を直截に書けるだけでなく，柔軟性に富んでいる．オブジェクトによる木構造の構築は，JavaScript を含む現代的な汎用プログラミング言語においては，ごく基本的な操作のひとつである．この基本的な構造に Web サイトの構成に関するあらゆる情報を組み込むことで，ソースファイル名と Web ページのパスの対応や，一部の階層の自動生成などを，自然な JavaScript コードによって表現できる．例えば，ページネーション機能も，データベースから情報の一覧を取得しページ単位に区切り整形する手続きを get 関数としてエクスポートするモジュールを作成し，それをモジュールの木構造の一部に組み込むことで，容易に達成できる．

3.2　動的インポートの静的解析

2 節で述べたとおり，Minissg では，各 Web ページからデータへの参照を，その Web ページを生成する実行パスの上で実行される import 文の副作用によって表現する．これを JavaScript の意味論の自然な拡張として実現するためには，意味論上は，生成プログラムの実行を HTML ファイルごとに分ける必要がある．とはいえ，生成プログラムの再起動を愚直に行うのは実行効率が悪い．本節では，再起動の繰り返しと同等の効果を一度の生成プログラムの実行で得るために行った工夫を説明する．

JavaScript のモジュールのインポートには，静的なインポートと動的なインポートの 2 種類が存在する．このうち，実行パスごとにインポートされるモジュールの集合を変化させることができるのは，動的インポートである．Minissg は生成プログラム中に含まれる動的インポートをフックし，その動的インポートの実行によって発現するデータの集合を，モジュールの木構造のパスごとに計算する．これを実現するために，Minissg は以下の 3 つのことを行う．

1. 生成プログラムのビルド時に Vite が計算したモジュール依存グラフから動的インポートによる依存関係を除去した有向グラフを作成し，各頂点を起点として連結成分を求める．これによって，各モジュールを動的インポートしたときに連れ立ってインポートされるデータファイルの集合が得られる．この情報を生成プログラムの実行前に計算しておく．
2. 生成プログラムに含まれる動的インポート式をそれぞれ，ビルド中に構文解析することで発見し，フックコードに構文的に置き換える．このフックコードは，本来の動的インポートを実行する前に，動的インポートしようとしているファイルの名前をキーとして 1 で得た情報を検索し，発現するデータの集合を得てから，それらのデータを現在の実行文脈において発現したデータの集合に追加する．

3. 生成プログラムが返した木構造を走査して HTML ファイル群を生成するとき，get 関数で生成が遅延されている箇所に遭遇するたびに，新しい実行文脈を Promise として生成する．現在走査中のパスにおいて発現したデータの集合は，この Promise にローカルな変数で管理する．木構造の走査が葉に到達したとき，この Promise にローカルな変数に含まれるデータを，その葉を内容とする HTML ファイルから参照されるデータとする．

この機構により，ユーザは JavaScript 標準の動的インポート機能を使うことで，Web ページ間のデータの分離を表現することができる．例えば，2 つの Web ページ `foo/index.html` と `bar/index.html` が，それぞれ独自のスタイルシート `foo.css` と `bar.css` を持つ状況を想定する．この状況に対応するコードを図 4 に示す．HTML ファイルからスタイルシートへの参照は，`foo/` を生成する `foo.js` に `import "./foo.css"` と書き，`bar/` を生成する `bar.js` に `import "./bar.css"` と書くことで表現できる（図 4(a) および (b)）．もし，トップレベルからこの 2 つの `.js` ファイルを静的にインポートすると（図 4(c)），トップレベルモジュールのロードと共に `foo.css` と `bar.css` のインポートが実行されてしまい，これら 2 つのスタイルシートが全ての HTML ファイルで発現する．一方，2 つの `.js` ファイルを動的にインポートした場合（図 4(d)），`foo/` への実行パスと `bar/` への実行パスで互いに独立に，発現するデータの集合が計算される．結果として，`foo/` でのみ `foo.css` が，`bar/` でのみ `bar.css` が，それぞれ有効となる（図 4(e)）．

3.3 ビルドプロセスの構成

これまでの説明をまとめると，Minissg の全体的な処理手順は以下のとおりである．

1. Vite を起動し生成プログラムをビルドする．
2. モジュール依存グラフを解析し，各ファイルを動的インポートしたときに発現するデータの集合を計算する．
3. 生成プログラムを実行しモジュールの木構造を得る．get 関数で遅延された部分木を生成しながらパスごとのデータ集合を計算する．
4. 確定した木構造から，データへの参照を含む HTML ファイル群を生成する．
5. 生成した HTML ファイル群をビルド対象として Vite をもう一度起動する．

従って，原理的には，この手順を逐次で実行するコー

(a) `foo.js`:

```
import "./foo.css";
export default "<html> ··· </html>";
```

(b) `bar.js`:

```
import "./bar.css";
export default "<html> ··· </html>";
```

(c) スタイルシートを分離しないトップレベル:

```
import * as foo from "./foo.js";
import * as bar from "./bar.js";
export const get = () => {
  "foo/": foo,
  "bar/": bar
};
```

(d) スタイルシートを分離するトップレベル:

```
const foo = () => import("./foo.js");
const bar = () => import("./bar.js");
export const get = () => {
  "foo/": { get: foo },
  "bar/": { get: bar }
};
```

(e) (c) が生成する `foo/index.html`:

```
<html><head>
  <link href="./foo.css" ···>
  <link href="./bar.css" ···>
</head> ···
```

(f) (d) が生成する `foo/index.html`:

```
<html><head>
  <link href="./foo.css" ···>
</head> ···
```

図 4　動的インポートによるスタイルシートの分離

ドを Minissg に組み込むだけで，Web サイト生成プロセスは完成する．

全体の手順を実装する上で著者が最も注意したのは，Vite がモジュールファイルを読み込む順番である．Vite プラグインの中には，ひとつのファイルの中にコードとデータの両方を含むファイルを処理するものが存在する．例えば，Vue.js の単一ファイルコンポーネントや Svelte コンポーネントなどは，その代表的な例である．これらのコンポーネントファイルは，HTML の断片と，その断片にのみ適用されるスタイルシートの両方を含む．これらのファイルを処理する Vite プラグインは，このファイルを解析するときに，ファイル内のスタイルシートを分離し，そのスタイルシートだけを含む仮想のファイルを生成する．この仮想のファイルの生成は副作用によって行われる．従って，コンポーネントファイルを読み込む前に，この仮想のスタイルシートファイルを読み込むことはできない．これは，ファイルを読み込む順番がビルド結果に影響することを意味する．

Minissg においては，このことは 2 回目の Vite の実行に影響を与える．この説明のため，例えば，図 5(a) に示す Svelte コンポーネントを用いて HTML ファイルを生成することを想定しよう．この `Hi.svelte` は，Vite の Svelte プラグインによって，図 5(b) のようなコードに変換される．この変換が行われると同時に，`Hi.svelte` のうち `<style>` で囲われた範囲に書かれている内容が，仮想のスタイルファイル `Hi.svelte.css`（実際にはもっと複雑な名前をしている）として切り出される（図 5(c)）．変換後のコードは，この仮想ファイルを `import` で参照している．Minissg はビルドされた生成プログラムを実行し，図 5(b) のコードの実行結果として，`Hi.svelte.css` への参照と Svelte コンポーネントの内容の両方を含む，図 5(d) のような HTML ファイルを生成する．この HTML ファイルを入力として，2 回目の Vite の実行が始まる．2 回目のビルドの目的は Web サイトのビルドであって生成プログラムのビルドではないので，生成プログラムの一部である `Hi.svelte` は読み込む必要がない．従って，仮想ファイル `Hi.svelte.css` は，2 回目の Vite の実行では作られない．しかしながら，入

(a) `Hi.svelte`:

```
<h1>Hi</h1>
<style> h1 { color: red } </style>
```

(b) `Hi.svelte` の JavaScript への変換結果:

```
const Hi = ···
export default Hi;
import "./Hi.svelte.css";
```

(c) 仮想のスタイルファイル `Hi.svelte.css`:

```
h1.svelte-zpl6q0 { color: red }
```

(d) Minissg が生成する HTML ファイル:

```
<html><head>
  <style href="./Hi.svelte.css" ···>
</head><body>
  <h1 class="svelte-zpl6q0">Hi</h1>
</body></html>
```

図 5　仮想スタイルファイルとその Minissg への影響

力の HTML ファイル（図 5(d)）には `Hi.svelte.css` への参照が含まれている．結果として，Vite はビルドに失敗し，`Hi.svelte.css` の読み込みエラーを報告する．

この問題を回避する方策として，Minissg では，Vite の 2 回目の実行であっても，生成された HTML ファイルを処理する前に，生成プログラムをビルド対象として全て読み込むこととした．これにより，生成された HTML ファイルの処理に必要な仮想ファイルをすべて生成してから，HTML ファイルの処理を行う．しかしながら，このビルドの結果として生成されるコードは，すでに実行が終わっている生成プログラムであり，最終的な Web サイトには含まれないコードである．この無駄なコードの読み込みは，モジュール依存グラフの構造に影響を与え，それゆえ最終的な Web サイトの最適化レベルにも影響を残す可能性がある．これを回避するため，2 回目の Vite の実行において，Minissg は以下のことを行う．

1. 1 回目の Vite の実行と同様に生成プログラムをビルドする．ただし，ファイルを読み込みプラグインによる変換が終わるたびに，その変換結

果のコードを `import` 文を残して全て破棄する．これにより，1 回目のビルドと同様の順番で各モジュールを読み込み，かつそれらのビルド結果が空になるようにする．

2. `import` 関係を追跡しながら，生成プログラムと最終的な Web サイトに含まれるモジュールの間で，モジュールの共有が行われていないかどうか検査する．もしモジュールの共有が検知された場合は，モジュールを複製して共有を避ける．これにより，モジュール依存グラフの中で生成プログラムが独立した連結成分となるようにする．

このようにして生成プログラムを分離・破棄することで，生成プログラムの読み込みに伴う最終的な Web サイトへの影響を最小限に抑える．

4 実用性のための機能拡張

Minissg を実用的な SSG とするためには，前節で述べた基本機能に加えて，実用上利便性が高いと思われる機能を提供することも重要である．とはいえ，機能を無闇に加えることは，コードサイズの増加を招き，メンテナンスのコストが嵩み，ソフトウェアの品質を悪くする．その実装の小ささを売りにする Minissg においてはなおさら，原理的には不要な機能を追加することは強く憚られる．本節では，Minissg が提供する便利な機能を，それを提供する理由とともに概説する．

4.1 クエリによる自動コード生成

React 等のコンポーネントシステムを用いて Web ページを組み立てた場合，組み立ての結果はコンポーネントとなるため，それをファイルに書き出すにはシリアライズを行う必要がある．Minissg は，HTML ファイルの書き出しのため，モジュールの木構造の葉には，ファイルにそのまま書き込める文字列やバイナリデータがあることを期待している．シリアライズ機能はどのコンポーネントシステムにも提供されており，マニュアルに従ってこの機能を利用するコードを書けば，原理的には，何の困難もなくユーザコードでシリアライズを達成できる．しかしながら，シリアライズ機能は通常の Web フロントエンド開発で用いられることは少なく，また Web サイトを構築するという本来の目的とは無関係である．そのような機能の使いこなしをユーザに要求してしまうと，コンポーネントシステムの利点を Minissg で享受できるのは，コンポーネントシステムに精通する一部の上級者に限られてしまう．

Minissg は，Vite のクエリパラメータを通じて，コンポーネントを簡便にシリアライズする機能を提供する．例えば，`./Foo.jsx` ファイルを，React のコンポーネントをデフォルトエクスポートするモジュールとする．このファイルを以下のように `?render` を付加してインポートすると，

```
import foo from "./Foo.jsx?render";
```

`./Foo.jsx` のコンポーネントをシリアライズした結果をインポートすることができる．

シリアライズと同様，煩雑だが実用上重要の機能のひとつに，パーシャルハイドレーションがある．Minissg では，`import` に `?hydrate` クエリを加えることで，パーシャルハイドレーションを簡便に実現できる．この機能は `?render` と同じ機構を用いて実現されている．

4.2 クライアント側コード

今日の Web サイトでは，Web ブラウザ上で実行する JavaScript コードが多用されている．Web サイトに埋め込む JavaScript コードを管理・操作する機構は，SSG において必須である．

Minissg では，生成プログラムが `import` するコードのうち，`?client` クエリを付加して `import` されたコードを，生成プログラムの一部ではなく，画像ファイルやスタイルシートと同様に，生成される HTML ファイルから参照されるデータとして取り扱う．コードへの参照を含む HTML ファイルは，2 回目の Vite の実行によって，Vite の標準機能によってビルドされ，その結果が最終的な Web サイトにそのまま組み込まれる．

4.3 木構造の文脈情報

HTML ファイルを生成するとき，モジュールの木構造の中におけるそのファイルの位置を知ることが

できると便利なことがある．その典型例のひとつは，ページネーションなど，ひとつのモジュールが複数の Web ページを互いの Web ページへのリンクを含む形で生成する場合である．このようなリンクを Web ページに含めるためには，生成される Web ページのパスに関する情報が必要である．このパス情報を決め打ちの定数として与えれば Web ページ間のリンクを生成することはできるが，この方法は可搬性や保守性に乏しい．Web ページのパスはモジュールの木構造から決まるので，各モジュールから自身の木構造内での文脈にアクセスできるのが望ましい．

Minissg では，木構造の生成を遅延する `get` 関数を呼び出すとき，その関数に至るまでに経由したパスを結合した文字列と，祖先のモジュールのリストを，引数として渡す．これにより，`get` 関数は木構造内での文脈情報を用いてページを生成することができる．これらの情報が不要であれば，JavaScript の仕様より，引数を受け取らない関数として `get` 関数を書けばよい．

4.4 開発サーバ

2 節で触れたとおり，開発サーバは Vite の主要機能のひとつであり，SSG においても有用な機能である．Vite の開発サーバは，リクエストを受け取った時に，レスポンスに必要なコードだけをビルドする．これによって，プロジェクトの規模にかかわらず，高速な開発サーバの起動を実現している．

Minissg で開発サーバを利用するために必要なことは，生成プログラムのオンデマンドなロードである．この実現のため，Minissg の開発サーバは，モジュールの木構造のうち，Web ブラウザからリクエストされた URI に関わる部分木だけを辿る．従って，その部分木に含まれるモジュールのみが，静的または動的なインポートによってロードされる．リクエストに対応する HTML ファイルが生成された後の処理は，Vite の標準機能にそのまま任せる．

5 Astro から Minissg への移行事例

Minissg の開発を始めた直接のきっかけは，著者が主宰する研究室の Web サイトを Astro [1] を用いて構築しようとして，不自由を感じたことにある．この不自由さの根源を確かめるために Astro で書いた Web サイトを Astro を使わずに書くことを試みた過程の中で，Minissg の設計や実装が固まった．本節ではこの書き換えの概要を報告する．

Astro は，最新の JavaScript フレームワークと同様の開発体験を提供しながら，デフォルトで JavaScript を含まない Web サイトを生成する Web フレームワークである．これを達成するため，Astro は，サーバ側でページ生成のために実行するコンポーネントを記述するための，JSX と類似した記法を持つ独自言語「Astro テンプレート」を導入する．

Astro と Minissg の主要な違いは，サーバ側コンポーネントの記述に独自言語を用いるか，JavaScript の汎用のコンポーネントシステムを使うかにある．Astro から Minissg に移行するにあたり必要なことの大部分は，Astro テンプレートを同等の JavaScript コードに書き換えることである．書き換えの結果，表現力やコード量に大きな変化がないのであれば，Minissg の当初の狙いである JavaScript のみによる Web サイトの生成が，Astro と同程度の簡便さで達成できたことになる．

書き換え前の Web サイトには以下の内容が含まれる．

- 現在の研究室メンバーおよび過去の学位論文のタイトルと概要の一覧．これはフロントマター付き Markdown ファイルの集合として作られた簡易なデータベースから自動生成される．
- 学会発表など研究室での出来事の一覧．出来事は，メンバーと同様，Markdown ファイルによる簡易データベースによって管理し，出来事の一覧をページネーションを行って生成する．
- 目次ボタンやアニメーション付きの折りたたみ要素など，Web ページ間で同じ実装が共有される汎用のコンポーネント．
- Preact と OpenPGP.js を用いて開発されたフロントエンド Web アプリケーション．

これらは，Markdown から HTML への単純なフォーマットの変換では実現できない，様々な計算と抽象化を伴う内容である．また，ほとんどのページは日本語

と英語の両方で書かれ，各 Web ページには言語の切り替えスイッチを置いている．なお，この Web サイトは制作途中であり，文章が未執筆の Web ページが多数存在する．

書き換え後の Web サイトは，書き換え前の Web サイトと外見上は同一である．一方，その内部構造はいくつかの点で書き換え前とは異なる．主要な差異は以下の 2 点である．1 点目は，Astro テンプレートに相当する機能として，コンポーネントの記述には Preact [7] を，スタイルシートの記述には Linaria [3] をそれぞれ採用したことである．2 点目は，ソースファイル名から Web ページのパスへの変換アルゴリズムを，日英対応が筆者の環境で明確になる独自のものに切り替えたことである．例えば，`/lab/ja/members/` に置かれる Web ページのソースは，Astro では `src/pages/lab/ja/members.astro` であるが，Minissg では `src/pages/lab/members.ja.tsx` とした．このような細かい調整は，Minissg が規約を持たないからこそ自在にできる芸当である．

書き換えは手作業で行った．JavaScript のモジュールや関数が自由に使える利点を享受するため，多くのファイルはスクラッチから書き直された．

書き換え前と書き換え後の両方で，ファイルを種類や役割ごとに分類し，各分類ごとの行数を集計した結果を表 1 に示す．アプリとデータベースは書き換え前後でほぼ同一であり，従って行数にもほぼ変化がない．また，Web ページの具体的な文章も，書き換え前後で同一である．従って，その他のページに分類される行数にも大きな変化は見られない．それ以外の分類に見られる主な差異は以下のとおりである．

- 設定においては，リンタやフォーマッタの設定ファイルに大きな差が見られた．書き換え前は，Astro テンプレートが独自言語であるため，これらのツールが Astro テンプレートを扱えるようにするための種々の設定が必要である．また，コミット前の自動検証においても，`.ts` ファイル，`.tsx` ファイル，および `.astro` ファイル間の参照を考慮する必要があり，複雑な設定を必要とした．書き換え後は，これらの設定はすべて不要である．

表 1　Astro から Minissg への書き換えによるソースファイルの行数の変化

分類	書き換え前	書き換え後
フレームワーク	Astro	Minissg
設定	401	292
プラグイン	1,601	1,639
ルーティング	27	32
ユーティリティ	354	467
共通コンポーネント	1,119	1,202
スタイルシート	566	604
アプリ	2,568	2,553
データベース	357	357
出来事ページ	306	304
メンバーページ	445	443
その他のページ	953	967
合計	8,697	8,860

- プラグインには総合的には行数の差がほとんどないが，書き換え前後それぞれにのみ含まれるものが存在する．例えば，書き換え前には，サーバ側コードとクライアント側コードで一部の定数を共有する機能を Astro に加えていた．書き換え後では，この機能は Minissg がクライアント側コードのサポートの一部として提供している．逆に，Astro は Markdown ファイルの見出し一覧を取得する機能を標準装備しているが，Minissg はこの機能を提供しないため，Rehype プラグインとしてこの機能を補っている．
- ルーティングについても行数の差がほとんどないが，その内容は全く異なる．Astro の規約では，ファイル名に `[` と `]` で囲ったパラメータを含めることでパスが可変であることを表し，ひとつのソースファイルから複数の Web ページを生成することができる．書き換え前では，複数の Web ページで共通するヘッダーやフッターを加えるために，この機能を利用している．書き換え後では，3.1 節の図 2 に示したような，`import.meta.glob` の結果をパスとモジュールの対応表に変換するモジュールをこの分類とした．

書き換え前は規約に基づくプログラミングをしているにもかかわらず，規約を使わないプログラミングとほぼ同じ行数を要したことは，興味深いところである．

- ユーティリティの分類には，フレームワークの機能を補うユーザコードが含まれる．Minissg のほうが機能が小さいため，より多くのユーティリティを必要とした．例えば，ページネーション機能や React のサスペンス機能への対応は，Minissg においてのみ補う必要があった．
- 共通コンポーネントの行数の増加の主な要因は，Astro コンポーネントが独自言語だからこその記述の簡潔さを汎用言語で書き切ったことにある．例えば，Astro コンポーネントは 1 つのファイルが暗黙に 1 つのコンポーネントとなるが，JavaScript では関数定義構文を用いてコンポーネントを関数として定義しエクスポートしなければならず，その分だけ記述量が増加する．
- スタイルシートの内容は書き換え前後でほぼ変わらない一方，書き換え後では `.css` ファイルと Linaria で定数を共有するための型定義が追加で必要となった．
- 出来事ページとメンバーページは，書き換え前には Astro のコンテンツコレクション機能を，書き換え後には Vite の `import.meta.glob` 機能を基盤としており，内容に大きな差異があるものの，最終的な行数にはほとんど差がない結果となった．

書き直しによる全体の行数の増加は 163 行に留まった．割合では約 2%の増加である．この増加の主な要因は，Astro にはあるが Minissg にはない機能をユーザコードで補ったことにあり，コンテンツそのものには目立った行数の増加は見られない．そのため，今後コンテンツが増えるに従って，この差は割合としてはますます小さくなるものと考えられる．また，増加した行数そのものも，10 万行を超える Astro と 1,100 行程度しかない Minissg の規模の差を含めれば，合理的な範囲に収まっている．

著者の主観に基づく個人の感想としては，コード量の増加を補って余りあるほどに，開発体験の改善を感じた．JavaScript や周辺ツールの機能をそのまま自由に使えるおかげで，最新のライブラリやツールを躊躇なく選ぶことができたこと，また，パスの生成など Astro では規約と衝突するこだわりを，Minissg では自然に実装することができたことが，開発体験に良い印象を持った主要な理由である．

今回の事例は，静的な Web サイトの生成でかつクライアント側コードを多用しなかったため，書き換え前に Astro の全ての機能を使用していたわけではない．例えば，Astro アイランド，ミドルウェア，バックエンドサービスなどを利用している場合は，異なる比較結果が現れたり，そもそも Minissg では容易に実現できないことに遭遇する可能性がある．一方，静的な Web サイトの生成に限るならば，今回の事例には，データベースからの情報取得，ページネーション，多言語サポートなど，Web サイトの構築において一般的な要件が多く含まれるため，今回の比較結果にはある程度の一般性があるものと思われる．この用途においては，Minissg は，小さいながらも既存のフルスタックフレームワークと比肩する利便性を有していると言える．

6 関連するソフトウェアとの比較

SSG や Web フレームワークには，様々な目的を掲げ様々な言語で書かれたものが多数存在しており，それら全てに言及することは不可能である．ここでは，JavaScript で書かれたものを中心に列挙・分類し，分類ごとに Minissg との比較を述べる．

Next.js [17]，Nuxt.js [11]，SvelteKit [16]，Fresh [4] などに代表される Web フレームワークは，特定のコンポーネントシステムと強く結びついている．この設計は，コンポーネントシステムの能力を最大限に引き出すことを容易にする．しかしながら，これらのフレームワークは一般に多くの規約を導入し，またフレームワークが下敷きとする機能の多くを抽象化して隠蔽する．そのため，フレームワークを用いて書かれたプログラムを読み書きするには，そのフレームワークに独自の知識を多く必要とする．残念なことに，フレームワークごとの独自の知識は一般に汎用性がなく，汎用言語に関する独自知識よりもかなり

早く腐敗する．結果として，フレームワークの採用は，そのフレームワークの盛衰にプロジェクトの命運を委ねることとなる．Minissg は，フレームワークの便利な機能が自動的に得られなくなることを犠牲に，プロジェクトをフレームワークから切り離し，言語やライブラリに関する汎用的な知識と技術による Web ページ制作を可能にする．

Astro [1] は，特定のコンポーネントシステムから独立した Web フレームワークである．独自のコンポーネント言語である Astro テンプレートを核として，異なるコンポーネントシステムの統合を実現している．Astro テンプレートは，本稿執筆時点では，Astro フレームワークでしか使用されてない独自言語である．Minissg は，特定の目的のために新しい言語を導入する Astro とは逆に，既存の言語を特定の目的に使用するアプローチを取る．もしこのアプローチに Web サイト制作上の障害があれば，それは特定のフレームワークの問題ではなく言語の問題であり，その解決は特定のフレームワークの利用者に留まらず，その言語の全ての利用者を利するはずである．

Gatsby [6] はシングルページアプリケーションを静的生成するためのフレームワークである．Gatsby の魅力のひとつは，その充実したプラグインシステムである．2,000 以上のプラグインが提供されており，ユーザがやりたいと思うほとんどのことが，プラグインを探してインストールするだけで実現する．このプラグインによるアプローチは，ユーザのあらゆる意図がプラグインの働きと合致していたならば，非常に有効に働く．しかしながら，もしユーザの意図と合致するプラグインがなかったり，プラグインがエラーを発生させたりしたときは，原因究明のためユーザーはシステムの奥底まで様々な調査をしなければならなくなる．また，Gatsby のプラグインは Gatsby でしか利用できず，その開発の努力は Gatsby コミュニティーに閉じてしまう．Minissg は，Web ページ制作を汎用言語の延長線上に置くことで，独自のシステムを作ることに起因するこれらの問題を根本的に回避する．一方，Gatsby がプラグインで実現しているあらゆることは，Minissg ではフレームワークから独立した汎用のライブラリを用いて実現することになる．

Eleventy [9] および Lume [12] は，Jekyll [8] の伝統を受け継ぐ，テンプレートから Web サイトを生成するソフトウェアである．これらのシステムの特徴のひとつは，ユーザがコードを書かないものとして設計されていることである．Web サイトの構成が複雑でなければ，例えば全ての Web ページがそれぞれ異なる単一のソースから生成されるならば，この設計は非常に有効である．その一方で，Web ページ間で共通する構造を共有するなど，何らかの制御構造や抽象構造が必要になったとき，ユーザーは汎用言語に比べて機能に乏しいテンプレート言語を駆使するか，システム固有のプラグインを書く必要に直面する．Minissg は，構造の共有こそが SSG の存在意義であり，従ってユーザがコードを書くのは避けられないことを前提とする．最新の汎用言語を Web サイト記述に充てることで，現時点における最高のコーディング体験を Web サイト制作に導入する．

minista [13] は，多量の機能を標準で提供する巨大なフレームワークは Web 制作において必ずしも必要でないことを示す好例のひとつである．minista は約 6,500 行の小さな Web フレームワークでありながら，その作者が必要とした全ての機能を含んでいる．この大きさのフレームワークが実用性に富んでいる理由のひとつは，JavaScript 言語そのものに加え，それを取り巻く汎用のライブラリやツールが十分に成熟しており，それらをただ組み合わせるだけで多くのニーズを満たすことができるからである．Minissg は minista とこの方向性を共有している．

Tropical [15] は，静的 Web サイトを生成するための Vite のテンプレートプロジェクトである．フルスタックフレームワークと同程度に強力でありながらより柔軟な Web 制作環境として，Vite を含む汎用ツールの組み合わせとその設定を提案している．Vite の直接利用によって Web 制作を実現するという点は，Minissg と同じである．その一方で，Minissg は Tropical とは異なり，ツールの組み合わせをユーザに提案しない．代わりに，汎用言語の制御下で Web サイトを制作するための簡便な基盤を提供する．ツールとライブラリの選定は，一般のアプリケーション開発と同様に，ユーザの選択に委ねられる．

vite-plugin-ssr [2] は，Vite を Web サイト制作に用いるための Vite プラグインであり，Minissg とは多くの目的や性質を共有する．両者は共に，特定の仕事だけを行い，どのフレームワークやツールからも独立で，実装が小さく単純でありながら，必要十分な機能を提供することを目指している．両者の違いは，Minissg はこれらの目的や性質を vite-plugin-ssr よりも強く追求していることにある．例えば，vite-plugin-ssr はファイルベースルーティングを採用するが，Minissg はルーティングについて何ら規約や仮定を置かない．その根拠は，ルーティングとは木構造の構築の一変種であり，木構造の構築はあらゆる高水準言語で簡単かつ簡潔に行うことができるのであるから，言語の中で簡単に書けることを言語の外に置く必要はない，というものである．言語の外に規約を置くことを徹底的に排除することが Minissg の設計原則であり，この原則が Minissg 自身を小さくしている．Minissg のコードサイズは vite-plugin-ssr の 1/8 以下である．

7 まとめ

本論文では，言語の外に規約を持たない SSG として Minissg を提案した．Minissg では Web サイト構築に関わるあらゆることを JavaScript のプログラムとして書き下す．ユーザは JavaScript が有する関数やモジュールなどの抽象機構を Web サイト制作に直接かつ自由に利用することができる．また，使用するライブラリの選択もユーザの自由である．Web 開発言語として成熟した JavaScript であれば，フレームワークの力を頼らなくても，Web 制作に必要十分な表現力を得ることができる．研究室の Web サイトを Astro から Minissg に書き直した事例を通じて，Minissg がフルスタックフレームワークに比肩する生産性を持っていることを確認した．

謝辞 本研究の一部は JSPS 科研費 19K11893 の助成を受けたものです．

参 考 文 献

[1] Astro Contributors: Astro, `https://astro.build`.
[2] Brillout, R.: vite-plugin-ssr, `https://vite-plugin-ssr.com`.
[3] Callstack: Linaria — zero-runtime CSS in JS library, `https://linaria.dev`.
[4] Casonato, L.: Fresh — The next-gen web framework, `https://fresh.deno.dev`.
[5] Compositor and Vercel, Inc.: Markdown for the component era — MDX, `https://mdxjs.com`.
[6] Gatsby, Inc.: The Fastest Frontend for the Headless Web — Gatsby, `https://www.gatsbyjs.com`.
[7] Jason Miller: Preact — Preact: Fast 3kb React alternative with the same ES6 API. Components & Virtual DOM., `https://preactjs.com`.
[8] Jekyll Core Team: Jekyll • Simple, blog-aware, static sites — Transform your plain text into static websites and blogs, `https://jekyllrb.com`.
[9] Leatherman, Z.: Eleventy, a simpler static site generator, `https://www.11ty.dev`.
[10] Meta Platforms, Inc.: JSX (Draft / August 4, 2022), `https://facebook.github.io/jsx/`.
[11] Nuxt Team: Nuxt: The Intuitive Web Framework, `https://nuxt.com`.
[12] Otero, O.: Lume, the static site generator for Deno - Lume, `https://lume.land`.
[13] QRANOKO: minista, `https://minista.qranoko.jp`.
[14] Shopify: Liquid template language, `https://shopify.github.io/liquid/`.
[15] Smithett, B.: Tropical — static site generator, `https://tropical.js.org`.
[16] Svelte Contributors: SvelteKit • Web development, streamlined, `https://kit.svelte.dev`.
[17] Vercel, Inc.: Next.js by Vercel - The React Framework, `https://nextjs.org`.
[18] You, E. and Vite Contributors: Vite — Next Generation Frontend Tooling, `https://vitejs.dev`.