WebAssemblyを用いたRSA暗号体験アプリの作成

福岡教育大学　藤本 光史
Mitsushi Fujimoto, University of Teacher Education Fukuoka

1　はじめに

Webブラウザに搭載されたJavaScriptエンジンの性能向上に伴い，ソフトウェアのプラットフォームとしてWebが注目されるようになってきた．筆者は [1] において，C言語で作成したコマンドラインアプリを WebAssembly を用いて Web アプリ化する手法を解説した．この手法を用いれば，計算エンジンとユーザインタフェース (UI) が分離された数学ソフトウェアを僅かな修正で Web アプリ化できる可能性がある．また，教育ソフトウェアの開発にも応用できると考える．

実際，筆者の所属大学での卒業研究 [2] において，本手法により RSA 暗号体験アプリを作成した．本稿はその概要の報告である．

2　JavaScript と WebAssembly

WebAssembly によるアプリ開発には WebAssembly だけでなく JavaScript の知識も必要である．これらがどのような技術なのか簡単に紹介する．

2.1　JavaScript

JavaScript は 1995 年に Web ブラウザの Netscape Navigator 2.0 に実装されたスクリプト言語であり，1997 年に Ecma International[1] により ECMAScript として標準化された．Web の動的コンテンツ作成ツールの Flash の台頭により一時低迷したが，2005 年にリリースされた Google Map が JavaScript を用いて開発されたことで再び脚光を浴びることになった[2]．

2008 年には Just-In-Time コンパイラを有した JavaScript エンジン V8 を搭載した Web ブラウザ Google Chrome がリリースされ，Web アプリのフロントエンドの基幹技術として定着した．また，2009 年にはサーバサイドの JavaScript 実行環境である Node.js がリリースされ，Web アプリのバックエンドでも利用されている．

現在の Web アプリの多くは JavaScript を基盤に開発されているため，Web ブラウザの開発元は競い合って JavaScript エンジンの性能向上に取り組んでいる．

[1]スイスのジュネーヴに本部を置く情報通信システム分野における国際的な標準化団体である．
[2]正確には，Ajax(Asynchronous JavaScript + XML) という技術を用いて開発された．

2.2 WebAssembly

WebAssembly（略称 Wasm）は Web ブラウザで実行される一種のバイナリコード（の仕様）である．これが策定された背景には，Web ブラウザで C/C++で書かれたコードをネイティブに実行したいという動機がある．JavaScript であれば Web ブラウザで実行できるので，C/C++で書かれたコードを JavaScript に変換すればよいわけである．しかし，JavaScript は動的型付け言語で，C/C++は静的型付け言語であるため，単純に変換しても最適化がうまくできず実行速度が遅くなってしまう。

そこで，JavaScript を静的型付け言語のように扱うためのサブセット asm.js が 2013 年に開発された．asm.js のコードは Ahead-Of-Time コンパイラによって高速に実行できるようになったが，そのコードは長大で読み込みと字句解析に時間がかかるという欠点があった．この欠点を解消するためにバイナリコードにしたものが WebAssembly である．WebAssembly は 2015 年から開発が開始され，2017 年に主要な Web ブラウザ（Google Chrome，Firefox，Safari，Edge）に搭載された．そして，2019 年 12 月に W3C が WebAssembly の仕様を勧告したことにより，WebAssembly は Web の標準技術になったのである．

3 WebAssembly で Web アプリ化されたソフトウェア

次の表は WebAssembly を用いて Web アプリ化されたソフトウェアの例である．

表 1: Wasm 化されたソフトウェア

種類	**アプリ名**	**Wasm 版リリース時期**
バーチャル 3D 地球儀	Google Earth	2020 年 2 月
Web 会議システム	Google Meet	2020 年 10 月
動画変換フレームワーク	FFmpeg	2021 年 8 月
将棋 AI	Shogimaru	2022 年 1 月
動画ストリーミング	Amazon Prime Video	2022 年 2 月
オフィススイート	LibreOffice	2022 年 2 月
Linux x86 仮想マシン	WebVM	2022 年 2 月

一方，C/C++で実装された数学ソフトウェアには以下のようなものがある．

表 2: 数学ソフトウェアの実装言語

C	**C++**	**C/C++**
R	Octave	Macaulay2
Coq(kernel)	Singular	MuPAD
Magma	CoCoA	
Risa/Asir	Giac	
Gap	Yacas	
PARI/GP		

これら既存数学ソフトウェアの Web アプリ化が今後の重要な課題である[3]．

[3]PARI/GP については既に WebAssembly 版が公開されている [3].

4　C 言語コマンドラインアプリの Web アプリ化

C 言語で開発されたコマンドラインアプリを WebAssembly を用いて Web アプリ化する手順は以下の通りである．

1. C 言語で作成されたコマンドラインアプリのソースを準備する．
2. Emscripten で C ソースをコンパイルする．その際，JavaScript 側から呼び出す C の関数名をオプションで渡す．
3. データの入出力の UI は HTML で，C の関数とのデータのやり取りは JavaScript で記述する．
4. Web ブラウザで HTML ファイルを開き，動作を確認する．

Emscripten[4] は C/C++のコードを Wasm バイナリに変換するコンパイラである．上記手順に従った詳しい解説とサンプルコードが [1] に記載されている．

5　WebAssembly を用いた RSA 暗号体験アプリ

RSA 暗号は初等整数論の良い教材である．ユークリッドの互除法の他に必要となるのはフェルマーの小定理のみであり，高校生にも十分理解可能である．旧学習指導要領の数学 A「整数の性質」は新学習指導要領では数学 A「数学と人間の活動」に移り，多くの高校で履修外となったが「課題学習」のプロジェクトとして最適なテーマの一つと考えられる．

そこで，筆者の研究室の卒業研究として，**鍵の生成→平文の暗号化→暗号文の復号化**を体験するためのアプリ開発を行うことにした．福岡教育大学の中等教育教員養成課程数学専攻では，数学専門科目の「コンピュータ」で C 言語を学んでいるのでアプリの計算エンジンには C 言語を，アプリの UI にはマルチプラットフォーム対応にするために Web を採用した．

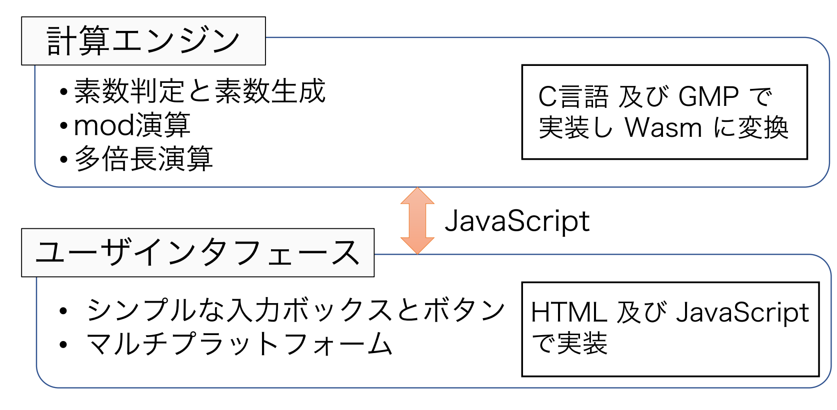

図 1: アプリの構成

また，巨大な整数を扱えるようにするために，多倍長演算ライブラリ GMP[5] を利用することにした．

6　アプリ開発プロセス

卒業研究の開始時に計画したアプリの開発プロセスは以下の通りである．

1. 計算エンジンの実装
   - 1-1. C 言語でプロトタイプ実装
   - 1-2. 各機能の関数化
   - 1-3. 暗号化と復号化のデータの受け渡しに文字列（`char *`）を利用
   - 1-4. 多倍長演算ライブラリ GMP の利用
2. 計算エンジンを Wasm へ変換
   - 2-1. WSL2 のインストール
   - 2-2. Emscripten 環境の構築と GMP のビルド
   - 2-3. C コマンドラインアプリの Wasm 化
3. UI の実装
   - 3-1. HTML と JavaScript の基礎を学習
   - 3-2. 入力ボックスとボタンの作成
   - 3-3. JavaScript からの C 関数の呼び出し
4. パフォーマンスの向上
   - 4-1. 素数判定でミラーラビンアルゴリズムを利用
   - 4-2. 復号化で中国剰余定理を利用
5. UI の機能向上
   - 5-1. ネットワークを利用したメッセージの送信・受信機能
   - 5-2. MathJax の利用

実際には上記の計画通りには進まず，卒業研究で完了できたのは上記 3 の「UI の実装」までであった．次節において，その詳細を説明する．

7　実際の開発の様子

7.1　計算エンジンの実装（`int` 型版）

RSA 暗号の鍵生成から暗号化や復号化を行うためには，素数判定，GCD，拡張ユークリッド互除法，ベキ剰余計算などが必要である．C 言語の初心者は関数化せず，これらすべてを `main` 関数に記述してしまうことが多いが，それでは UI とのデータのやり取りが困難になるので，「`main` 関数は整数や文字列の入出力だけにして，素数判定・鍵生成・暗号化・復号化などは関数化するように」という助言を行った．また，計算エンジンと UI のデータの受け渡しは文字列で行う必要があるので，各関数の戻り値の型を `char *` にするよう指導した．

最初に出来上がったプログラムは整数型に `int` を用い，平文を一文字ずつ暗号化するようになっていた．後で GMP を導入することから整数型は `int` のままで良しとした

が,「これでは暗号文から平文が簡単に類推できてしまうので，２文字ずつまとめて数値化してから暗号化するように変更しましょう」と助言した．

int 型版の計算エンジンのソースファイル rsa.c 内の主な関数は以下である．

表 3: 計算エンジンの C 関数 (int 型版)

関数のプロトタイプ	機能
int nextprime(int p)	p を超える素数を生成
int key_m(int p, int q)	素数 p,q から公開鍵 m を生成
int key_e(int p, int q)	素数 p,q から公開鍵 e を生成
int key_d(int p, int q)	素数 p,q から秘密鍵 d を生成
char *text_numstr(char *txt, int n)	平文 txt を n 文字ずつ数値化
char *encrypt(char *txt, int n, int m, int e)	平文 txt を公開鍵 m,e で暗号化
char *decrypt(char *en_txt, int m, int d)	暗号文 en_txt を秘密鍵 d で復号化

平文の入力には半角英数記号を用いて，ASCII コードで数値化している．

7.2 計算エンジンを Wasm へ変換

ここまでは Windows 環境（Tiny C Compiler）で開発していたが，Emscripten の導入をスムースに行うために Linux 環境で行うよう指導した．WSL を用いれば簡単に Windows 上に Linux 環境を構築可能[4]であるが，コマンドライン操作に不慣れな学生には Emscripten 導入後の「環境変数の設定」についてフォローが必要であった．

Emscripten でソースファイル rsa.c を Wasm バイナリに変換するには以下のようにすればよい[5]．

```
$ source ~/emsdk/emsdk_env.sh
$ emcc rsa.c -o rsa.js -s WASM=1 -s "EXPORTED_FUNCTIONS=['_nextprime',
 '_key_m', '_key_e', '_key_d', '_text_numstr', '_encrypt', '_decrypt_numstr',
 '_decrypt']" -s "EXPORTED_RUNTIME_METHODS=['ccall']"
```

これによって，Wasm バイナリファイル rsa.wasm と JavaScript ファイル rsa.js が生成される．rsa.js はグルーコードであり，rsa.wasm をロードするために利用される．

7.3 UI の実装

アプリのユーザインタフェースを HTML と JavaScript で作成する．入力ボックスを<input>タグで，ボタンを<button>タグで，計算結果を出力する場所を<div>タグで記述する．そして，前節で用意したグルーコードの rsa.js を<script>タグの src 属性で指定し，さらに<script>タグ内で C 関数とのデータのやり取りを記述する．

以下に作成した HTML ファイル rsa.html の一部を示す．

[4] [1] の第 4 章に WSL の導入の詳しい解説がある．

[5] EXPORTED_FUNCTIONS や EXPORTED_RUNTIME_METHODS などのオプションの詳細については [1] を参照のこと．

Listing 1: rsa.html

```
<label>p = <input type="number" size="10" id="p"/></label>
<button type="button" id="run1">素数判定</button>
<div id="result1"></div>
...
...
<script src="rsa.js"></script>
<script>
...
document.getElementById('run1').addEventListener('click', () => {
  var p = document.getElementById('p');
  p1 = Module.ccall('nextprime', 'number', ['number'], [p.value]);
  if(p.value == p1){
    document.getElementById('result1').innerHTML
    = "素数です <b>p:" + p1 + "</b>";
  }else{
    document.getElementById('result1').innerHTML
    = "補正しました <b>p:" + p1 + "</b>";
  }
});
...
</script>
```

`id="run1"`の「素数判定」ボタンがクリックされたら，`id="p"`の入力ボックスに入力された整数が変数 p に保存され，その値が `ccall` により C 言語の `nextprime` 関数に渡され，結果が `id="result1"`の`<div>`タグに出力される[6]．

鍵の生成に必要となる素数 p, q については，入力された自然数を素数判定し，素数でない場合は「その数を超える最小の素数」に補正するようにした．また，公開鍵と秘密鍵の入力ボックスにはデフォルト値として，上で生成したものを設定するようにした．

図 2: ２つの素数から鍵を生成　　図 3: 暗号化と復号化

ユーザインタフェースの動作を確認するには，ローカル Web サーバを起動してから Web ブラウザで `rsa.html` を開けばよい．

---

[6] `ccall` の仕様の詳細については [1] を参照のこと．

7.4 計算エンジンの実装（GMP 版）

ここまでの計算エンジンでは平文を 2 文字毎にまとめて数値化し暗号化していたが，3 文字以上でまとめると暗号化に失敗してしまう．また，素数 p, q についても 5 桁程度までしか対応していない[7]．これは扱う整数の桁数が `int` 型の範囲を超えたことが原因である．この制限を回避するために多倍長計算ライブラリ GMP を利用する．

WSL に GMP をインストールするには `apt install libgmp-dev` を実行する方法と，Web からソースコードを入手してビルドする方法がある．ここでは，Emscripten 用の GMP も作成する必要があるため後者を採用する．まず，(Emscripten 用でない通常の) GMP をインストールする．

```
$ curl -O https://gmplib.org/download/gmp/gmp-6.2.1.tar.bz2
$ tar xf gmp-6.2.1.tar.bz2
$ cd gmp-6.2.1
$ ./configure --prefix=/usr/local
$ make
$ sudo make install
```

GMP を利用するために，ソースコード `rsa_gmp.c` の先頭に`#include <gmp.h>`を追加し，必要に応じて `int` 型を GMP の `mpz_t` 型に変更する．変更例として，`int` 型版の `rsa.c` と GMP 版の `rsa_gmp.c` の素数判定関数（試し割り法）のコードを示す．

Listing 2: rsa.c の primecheck 関数

```
int primecheck(int p){
  int i = 2;
  while(i <= sqrt(p)){
    if( p % i != 0 ){
      i++;
    }else{
      break;
    }
  }
  if(i > sqrt(p)){
    return 1;
  }else{
    return -1;
  }
}
```

Listing 3: rsa_gmp.c の primecheck 関数

```
int primecheck(mpz_t p){
  mpz_t i, s, r;
  mpz_init(i); mpz_init(s); mpz_init(r);
  mpz_set_ui(i, 2);
```

[7] $m = pq$ が 32 ビットで収まるような p, q でないといけない．

```
  mpz_sqrt(s, p);
  while( mpz_cmp(s, i) > 0 ){
    mpz_tdiv_r(r, p, i);
    if( mpz_cmp_ui(r, 0) != 0 ){
      mpz_add_ui(i, i, 1);
    }else{
      break;
    }
  }
  if( mpz_cmp(s, i) == 0 ){
    mpz_clear(i); mpz_clear(s); mpz_clear(r);
    return 1;
  }else{
    mpz_clear(i); mpz_clear(s); mpz_clear(r);
    return -1;
  }
}
```

この他に nextprime，key_m，key_e，key_d 関数の戻り値の型が char * に変更され，text_numstr，encrypt，decrypt 関数に渡される鍵の型が char * に変更される．また，GMP には GCD，拡張ユークリッド互除法，ベキ剰余計算の関数が提供されているので，それらを利用するように修正する．

こうして出来上がった GMP 版計算エンジンを Wasm バイナリにするために，GMP 自体を Emscripten でビルドする．Emscripten 用 GMP は ~/opt ディレクトリに保存することにする．

```
$ cd
$ mkdir opt
$ cd gmp-6.2.1
$ make distclean
$ emconfigure ./configure --host=wasm32-unknown-emscripten
  --prefix=${HOME}/opt
$ emmake make
$ emmake make install
```

そして，以下のように Emscripten でソースファイル rsa_gmp.c をビルドする[8]．

```
$ emcc rsa_gmp.c -o rsa_gmp.js -I${HOME}/opt/include -static
  -L${HOME}/opt/lib -lgmp -s WASM=1 -s "EXPORTED_FUNCTIONS=['_nextprime_str',
  '_key_m','_key_e', '_key_d', '_climit', '_text_numstr', '_encrypt',
  '_decrypt_numstr', '_decrypt']" -s "EXPORTED_RUNTIME_METHODS=['ccall']"
  -s INITIAL_MEMORY=512MB -s ALLOW_MEMORY_GROWTH=1
```

この GMP 版計算エンジンを利用するために，rsa.html の<script src="rsa.js">の箇所を<script src="rsa_gmp.js">に変更する．

[8]現在の WebAssembly は動的リンクへの対応が不完全であるため，-static オプションを付けて GMP を静的リンクする必要がある．

8 Emscriptenのモジュール化の利用

「UIの実装」まで完了した段階のアプリの動作を確認すると，入力ボックスに入力する整数が10桁程度であれば問題なく動作するが，それ以上では無反応になってしまう．これは素数判定に試し割り法を使用していることが原因である．

RSA暗号では素数p, qに512〜1024ビットの整数を利用するのが一般的であるので，体験アプリとして10進100桁程度の整数が入力された場合でも問題なく動作するようにすべきである．そのためには高速な素数判定アルゴリズムを使用する必要がある．ここでは，ミラー・ラビン法を利用する．

計算エンジンのC言語ソースコード`rsa_gmp.c`の`primecheck`関数を修正するのではなく，ミラー・ラビン法を実装した`pseudoprime_str`関数が記述されている別ファイル`rsa_gmp_miller.c`を準備し利用する．

複数のC言語ソースコードを混在して利用するにはEmscriptenのモジュール化オプションが便利である．モジュール化を使うには，`rsa_gmp_miller.c`を以下のようにEmscriptenでビルドする．

```
$ emcc rsa_gmp_miller.c -o rsa_gmp_miller.js -I${HOME}/opt/include -static
  -L${HOME}/opt/lib -lgmp -s WASM=1
  -s "EXPORTED_FUNCTIONS=['_pseudoprime_str']"
  -s "EXPORTED_RUNTIME_METHODS=['ccall']" -s INITIAL_MEMORY=512MB
  -s ALLOW_MEMORY_GROWTH=1 -s MODULARIZE=1 -s EXPORT_NAME=miller
```

`MODULARIZE=1`がモジュール化を有効にするオプションであり，`EXPORT_NAME=miller`によって，Moduleオブジェクトを生成するための関数名`miller`を設定する．

そして，以下のように`rsa.html`に追加・修正を施す．

Listing 4: rsa.html

```
...
<script src="rsa_gmp.js"></script>
<script src="rsa_gmp_miller.js"></script> //追加
<script>
...
document.getElementById('run1').addEventListener('click', () => {
  miller().then(mod => { //追加
    var p = document.getElementById('p');
    p1 = mod.ccall('pseudoprime_str','string',['string'],[p.value]); //修正
    if(p.value == p1){
      document.getElementById('result1').innerHTML
      = "素数です <b>p:" + p1 + "</b>";
    }else{
      document.getElementById('result1').innerHTML
      = "補正しました <b>p:" + p1 + "</b>";
    }
  }); //追加
});
...
```

```
</script>
```

以上の修正により，rsa_gmp.wasm と rsa_gmp_miller.wasm の 2 つの Wasm バイナリを同時に利用できるようになり，入力ボックスに 100 桁程度の整数を入力した場合も以下のように問題なく動作するようになる．

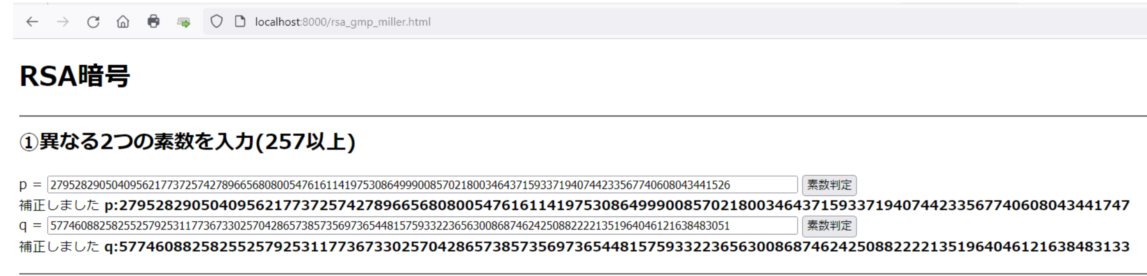

図 4: 111 桁の整数も入力可能

9　おわりに

本稿で紹介した RSA 暗号体験アプリは，計算エンジンは C 言語で，UI は HTML と JavaScript で作成された Web アプリである．本手法を用いることにより，既存の数学ソフトウェアのうち C/C++ で開発されたものを Web アプリ化できる可能性がある．

Web アプリ化することにより，他の JavaScript ライブラリや Web 技術との連携が容易になるという利点がある．例えば，MathJax を用いて計算結果をグラフィカルな数式で表示したり，React を用いることでよりリッチな GUI を作成することが可能になる．

また，数学者が論文作成の過程で作成した C/C++アプリは公開されないことが多いが，本手法を用いれば比較的容易にそれらを Web アプリ化できると思われる．そうやって論文で使用されたアプリが Web アプリとして公開されるようになれば，論文の査読者や読者が容易に検証することが可能になるかもしれない．

謝辞

本研究は JSPS 科研費 21K12157 の助成を受けている．

参考文献

[1] 高山信毅, 野呂正行, 小原功任, 藤本光史: 数学ソフトウェアの作り方, 共立出版, 2022.

[2] 山下蓮: 初等整数論を題材にした Web 教材の作成, 福岡教育大学卒業論文, 2023.

[3] Run PARI/GP in your browser, https://pari.math.u-bordeaux.fr/gpwasm.html

[4] Emscripten, https://emscripten.org/

[5] GMP – The GNU Multiple Precision Arithmetic Library, https://gmplib.org/