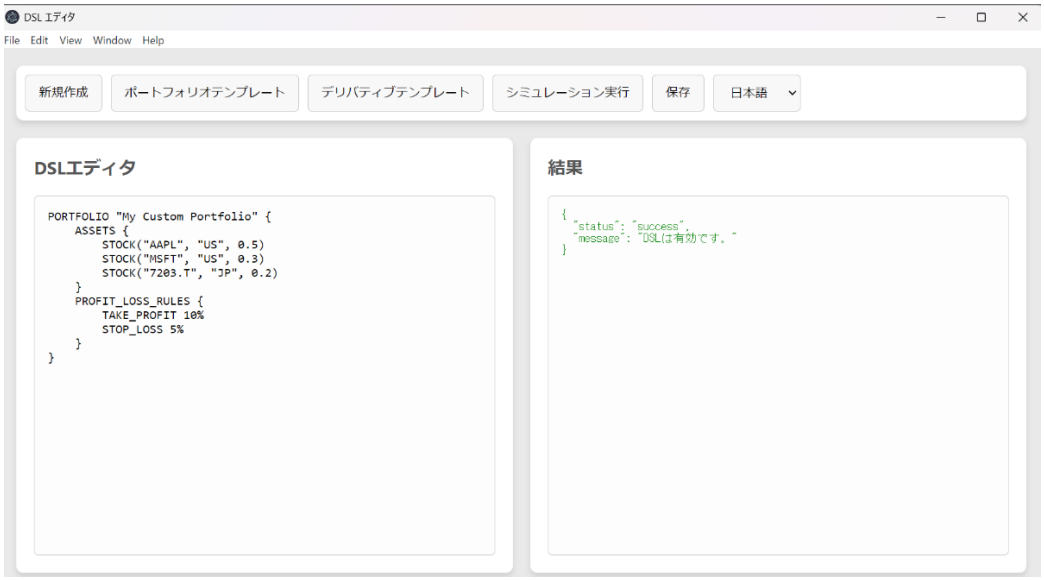



```
const { ipcRenderer } = require('electron');

document.addEventListener('DOMContentLoaded', () => {

    // 翻訳テキストをオブジェクトとして定義
    const translations = {
        'ja': {
            'newBtn': '新規作成',
            'saveBtn': '保存',
            'templateBtn': 'テンプレートから作成',
            'runBtn': 'シミュレーション実行',
            'editorTitle': 'DSL エディタ',
            'editorPlaceholder': 'ここに DSL を記述してください...',
            'resultTitle': '結果',
```

```
            'resultPlaceholder': 'シミュレーション結果やエラーメッセージがここに表示されます。',
            'saveSuccess': 'ファイルが正常に保存されました。',
            'saveError': 'ファイルの保存中にエラーが発生しました。'
        },
        'en': {
            'newBtn': 'New',
            'saveBtn': 'Save',
            'templateBtn': 'Create from Template',
            'runBtn': 'Run Simulation',
            'editorTitle': 'DSL Editor',
            'editorPlaceholder': 'Write your DSL here...',
            'resultTitle': 'Result',
            'resultPlaceholder': 'Simulation results and error messages will be displayed here.',
            'saveSuccess': 'File saved successfully.',
            'saveError': 'An error occurred while saving the file.'
        },
        'ko': {
            'newBtn': '새로 만들기',
            'saveBtn': '저장',
            'templateBtn': '템플릿으로 만들기',
            'runBtn': '시뮬레이션 실행',
            'editorTitle': 'DSL 편집기',
            'editorPlaceholder': '여기에 DSL을 작성하세요...',
```

```
        'resultTitle': '결과',
        'resultPlaceholder': '시뮬레이션 결과 및 오류 메시지가 여기에 표시됩니다。',
        'saveSuccess': '파일이 정상적으로 저장되었습니다.',
        'saveError': '파일 저장 중 오류가 발생했습니다.'
    },
    'zh': {
        'newBtn': '新建',
        'saveBtn': '保存',
        'templateBtn': '从模板创建',
        'runBtn': '运行模拟',
        'editorTitle': 'DSL 编辑器',
        'editorPlaceholder': '在此处编写您的 DSL...',
        'resultTitle': '结果',
        'resultPlaceholder': '模拟结果和错误消息将显示在此处。',
        'saveSuccess': '文件保存成功。',
        'saveError': '保存文件时发生错误。'
    }
};

// UI のテキストを更新する関数
function updateContent(lang) {
    const t = translations[lang];
    if (!t) return;
```

```
        document.getElementById('newBtn').innerText = t.newBtn;
        document.getElementById('saveBtn').innerText = t.saveBtn;
        document.getElementById('templateBtn').innerText = t.templateBtn;
        document.getElementById('runBtn').innerText = t.runBtn;
        document.querySelector('.editor-panel h2').innerText = t.editorTitle;
        document.querySelector('.result-panel h2').innerText = t.resultTitle;
        document.getElementById('dslInput').placeholder = t.editorPlaceholder;
        document.getElementById('resultOutput').querySelector('p').innerText =
t.resultPlaceholder;
    }

    const dslInput = document.getElementById('dslInput');
    const templateBtn = document.getElementById('templateBtn');
    const newBtn = document.getElementById('newBtn');
    const languageSelector = document.getElementById('languageSelector');
    const saveBtn = document.getElementById('saveBtn');
    const runBtn = document.getElementById('runBtn');
    const resultOutput = document.getElementById('resultOutput');
    let currentLang = 'ja';

    // ページロード時にデフォルト言語（日本語）を設定
    updateContent(currentLang);

    // 言語セレクターが変更されたときのイベントリスナー
```

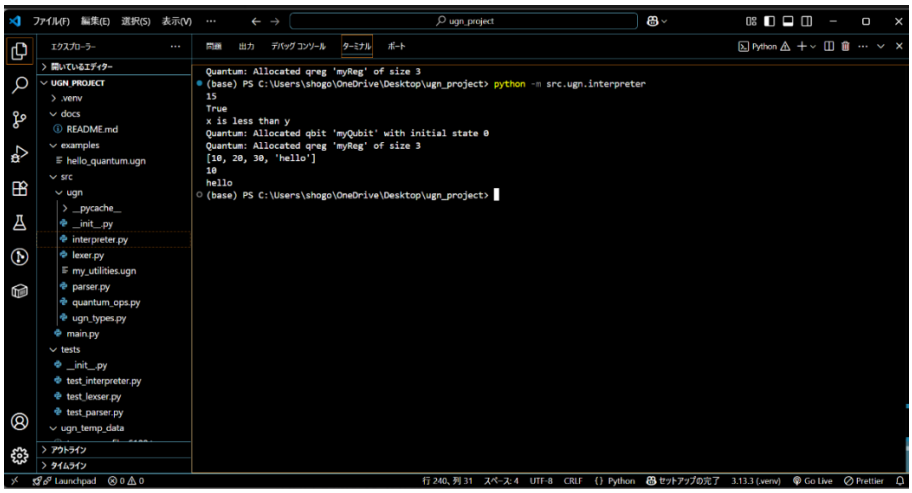
```
    languageSelector.addEventListener('change', (e) => {
        currentLang = e.target.value;
        updateContent(currentLang);
    });

    const dslTemplate = `DERIVATIVE "My Custom Product" {
    ASSETS {
        STOCK("AAPL", "US", 0.5)
        STOCK("MSFT", "US", 0.3)
        STOCK("7203.T", "JP", 0.2)
    }
    LEVERAGE 3.0
    PROFIT_LOSS_RULES {
        TAKE_PROFIT 10%
        STOP_LOSS 5%
    }
}`;

    templateBtn.addEventListener('click', () => {
        dslInput.value = dslTemplate;
    });

    newBtn.addEventListener('click', () => {
        dslInput.value = '';
```

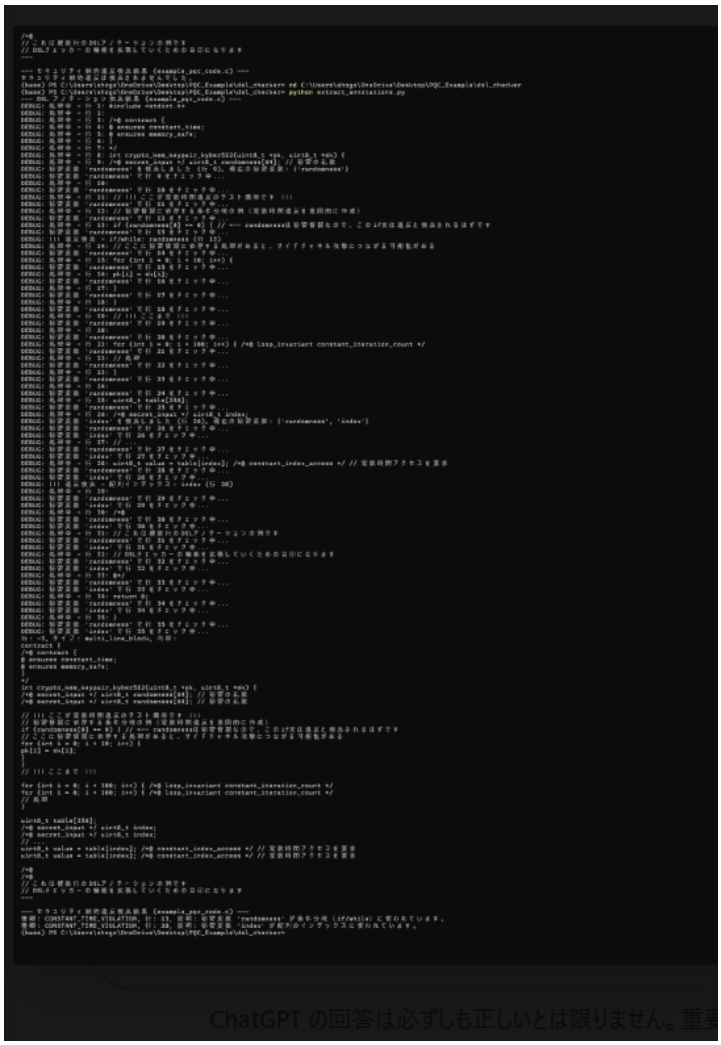
```
    });

    // 保存ボタンのイベントリスナー
    saveBtn.addEventListener('click', () => {
        const content = dslInput.value;
        ipcRenderer.send('save-file', content);
    });

    // 保存結果をメインプロセスから受け取る
    ipcRenderer.on('save-result', (event, response) => {
        const t = translations[currentLang];
        if (response.status === 'success') {
            resultOutput.innerHTML = `<p style="color: green;">${t.saveSuccess}</p>`;
        } else {
            resultOutput.innerHTML = `<p style="color: red;">${t.saveError}:
${response.message}</p>`;
        }
    });

    // バリデーションを実行する非同期関数
    async function validateDsl(dslCode) {
        const apiUrl = 'http://127.0.0.1:8000/validate_dsl';

        try {
```

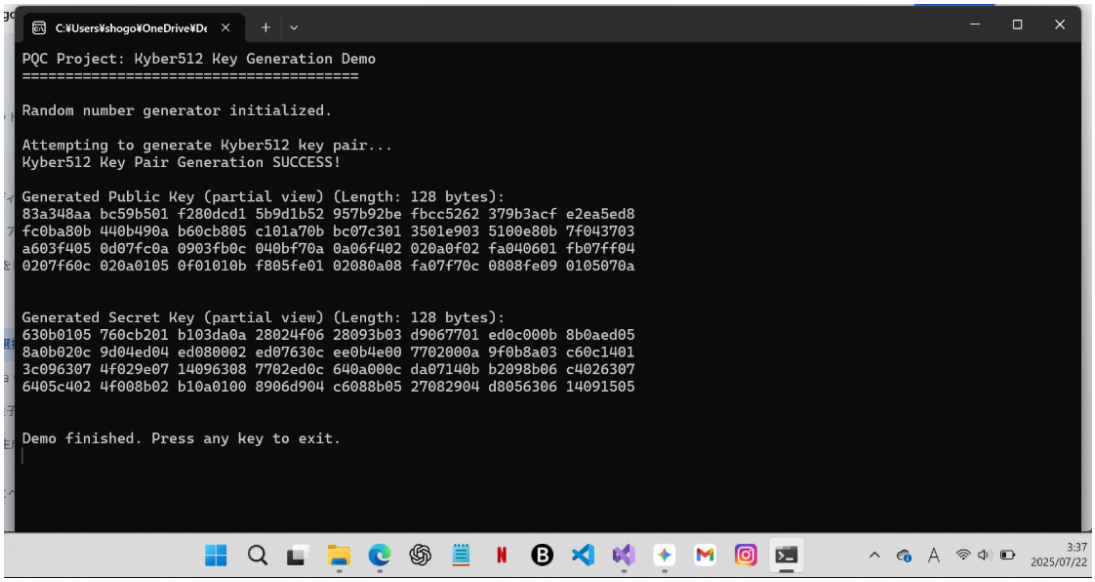
```
            const response = await fetch(apiUrl, {
                method: 'POST',
                headers: {
                    'Content-Type': 'application/json'
                },
                body: JSON.stringify({ dsl_code: dslCode })
            });

            const result = await response.json();
            const formattedResult = JSON.stringify(result, null, 2);

            if (response.ok) {
                resultOutput.innerHTML = `<pre style="color: green;">${formattedResult}</pre>`;
            } else {
                resultOutput.innerHTML = `<pre style="color: red;">${formattedResult}</pre>`;
            }
        } catch (error) {
            resultOutput.innerHTML = `<pre style="color: red;">バックエンドへの接続に失敗しました: ${error.message}</pre>`;
        }
    }

    // デバウンス関数
```

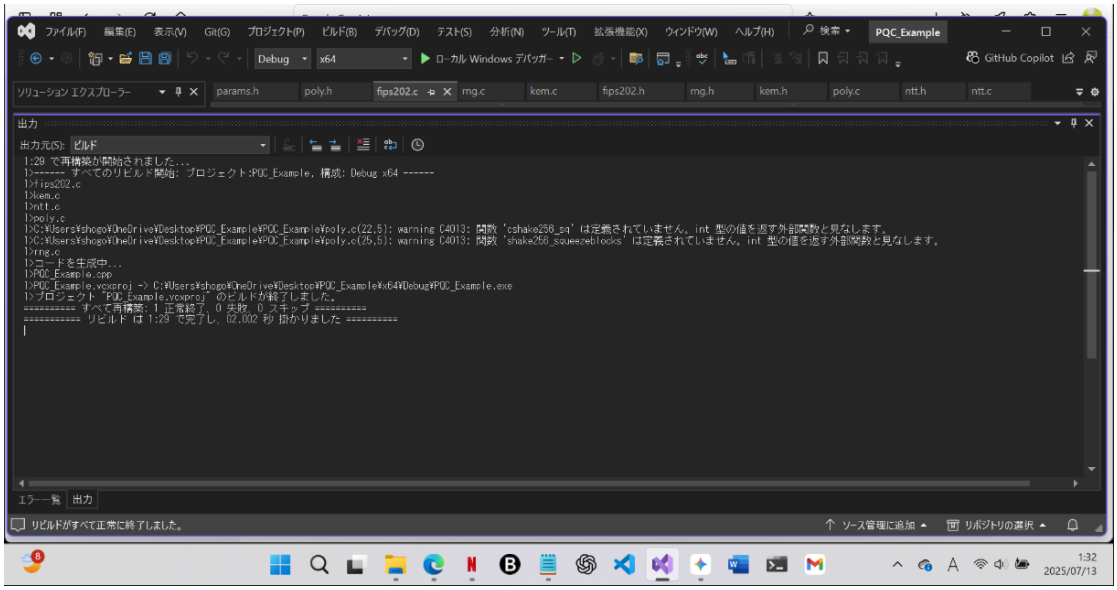
```
    const debounce = (func, delay) => {
        let timeoutId;
        return (...args) => {
            clearTimeout(timeoutId);
            timeoutId = setTimeout(() => func(...args), delay);
        };
    };

    // リアルタイムバリデーション（入力が停止してから 500ms 後に実行）
    const debouncedValidate = debounce(validateDsl, 500);

    dslInput.addEventListener('input', (e) => {
        const dslCode = e.target.value;
        if (dslCode.trim() === '') {
            resultOutput.innerHTML = `<p>シミュレーション結果やエラーメッセージがここに表示されます。</p>`;
            return;
        }
        resultOutput.innerHTML = `<p style="color: gray;">入力を確認中です...</p>`;
        debouncedValidate(dslCode);
    });

    // 「シミュレーション実行」ボタンのイベントリスナー
    runBtn.addEventListener('click', () => {
```

```
        const dslCode = dslInput.value;

        resultOutput.innerHTML = `<p style="color: gray;">シミュレーションを実行中です...</p>`;

        validateDsl(dslCode);

    });

});
```









【#include "kem.h"】

```
#include "poly.h" // poly_t, polyvec_t の定義のため
#include "rng.h"
#include "fips202.h"
#include "ntt.h"
#include <string.h>

// poly_add: 2 つの多項式を加算（r = a + b）
void poly_add(poly_t* r, const poly_t* a, const poly_t* b) {
    for (int i = 0; i < KYBER_N; i++) {
        r->coeffs[i] = a->coeffs[i] + b->coeffs[i];
        if (r->coeffs[i] >= KYBER_Q) {
            r->coeffs[i] -= KYBER_Q;
        }
    }
}

// poly_sub: 2 つの多項式を減算（r = a - b）
void poly_sub(poly_t* r, const poly_t* a, const poly_t* b) {
    for (int i = 0; i < KYBER_N; i++) {
        r->coeffs[i] = a->coeffs[i] - b->coeffs[i];
        if (r->coeffs[i] < 0) {
            r->coeffs[i] += KYBER_Q;
        }
    }
```

```
}

// poly_basemul: NTT ドメインでの 2 つの多項式の乗算（r = a * b）
// これは ntt.c の poly_basemul_montgomery を呼び出すラッパー
void poly_basemul(poly_t* r, const poly_t* a, const poly_t* b) {
    poly_basemul_montgomery(r, a, b);
}

// polyvec_basemul_montgomery_acc:
// r += A * b (NTT ドメイン)
// r: poly_t* (結果の多項式)
// a: polyvec_t* (多項式ベクトル、行)
// b: polyvec_t* (多項式ベクトル、列)
void polyvec_basemul_montgomery_acc(poly_t* r, const polyvec_t* a, const polyvec_t* b) {
    poly_t t; // 一時的な多項式

    // r をゼロで初期化
    for (int k = 0; k < KYBER_N; ++k) {
        r->coeffs[k] = 0;
    }

    for (int i = 0; i < KYBER_K; i++) {
        poly_basemul(&t, &a->vec[i], &b->vec[i]); // t = a_i * b_i (NTT ドメイン)
        poly_add(r, r, &t); // r += t
```

```
    }
}

// 鍵生成関数
// pk: 公開鍵（バイト列）
// sk: 秘密鍵（バイト列）
int crypto_kem_keypair_kyber512(uint8_t* pk, uint8_t* sk) {
    polyvec_t A[KYBER_K]; // KxK の多項式ベクトル行列
    polyvec_t sk_polyvec; // 秘密鍵の多項式ベクトル s
    polyvec_t e_polyvec;  // エラー多項式ベクトル e
    polyvec_t pk_polyvec; // 公開鍵の多項式ベクトル t

    uint8_t rho[KYBER_SYMMETRIC_BYTES]; // 行列 A の生成シード
    uint8_t tr[KYBER_SYMMETRIC_BYTES];  // 秘密鍵の生成シード
    uint8_t randomness[2 * KYBER_SYMMETRIC_BYTES]; // 乱数ジェネレータからの出力

    // 1. 乱数シードの生成
    randombytes(randomness, 2 * KYBER_SYMMETRIC_BYTES);
    memcpy(rho, randomness, KYBER_SYMMETRIC_BYTES);
    memcpy(tr, randomness + KYBER_SYMMETRIC_BYTES,
KYBER_SYMMETRIC_BYTES);

    // 2. 行列 A_hat の生成（NTT ドメイン）
    gen_matrix(A, rho, 0); // poly.c の gen_matrix を呼び出す
```

```
    // 3. 秘密鍵 s とエラー多項式 e の生成（NTT ドメイン）
    for (int i = 0; i < KYBER_K; i++) {
        poly_uniform(&sk_polyvec.vec[i], tr, i);                // s_i
        poly_uniform(&e_polyvec.vec[i], tr, KYBER_K + i);  // e_i

        // 生成された s_i と e_i を NTT ドメインに変換
        poly_to_ntt(&sk_polyvec.vec[i]);
        poly_to_ntt(&e_polyvec.vec[i]);
    }

    // 4. 公開鍵 t_hat の計算: t_hat = A_hat * s_hat + e_hat
    for (int i = 0; i < KYBER_K; i++) { // t_hat の各要素 t_hat_i を計算
        polyvec_basemul_montgomery_acc(&pk_polyvec.vec[i], &A[i], &sk_polyvec); // t_hat_i = sum(A_i,j * s_j)
        poly_add(&pk_polyvec.vec[i], &pk_polyvec.vec[i], &e_polyvec.vec[i]);      // t_hat_i += e_i
    }

    // 5. 公開鍵と秘密鍵のバイト列への変換と結合

    // 公開鍵の構築
    memcpy(pk, rho, KYBER_SYMMETRIC_BYTES); // pk の先頭に rho をコピー
    for (int i = 0; i < KYBER_K; i++) {
```

```
        poly_from_ntt(&pk_polyvec.vec[i]); // NTT ドメインから通常ドメインに戻す
        poly_compress(&pk[KYBER_SYMMETRIC_BYTES + i * KYBER_POLYCOMPRESSED_BYTES], &pk_polyvec.vec[i]);
    }

    // 秘密鍵の構築
    for (int i = 0; i < KYBER_K; i++) {
        poly_from_ntt(&sk_polyvec.vec[i]); // NTT ドメインから通常ドメインに戻す
        poly_compress(&sk[i * KYBER_POLYCOMPRESSED_BYTES], &sk_polyvec.vec[i]);
    }
    memcpy(&sk[KYBER_K * KYBER_POLYCOMPRESSED_BYTES], tr, KYBER_SYMMETRIC_BYTES); // 秘密鍵に tr を追加

    return 0; // 成功
}

// カプセル化関数
// c: 暗号文（バイト列）
// k: 共通鍵（バイト列）
// pk: 公開鍵（バイト列）
int crypto_kem_encapsulate_kyber512(uint8_t* c, uint8_t* k, const uint8_t* pk) {
    polyvec_t pk_polyvec_decomp; // 展開された公開鍵の多項式ベクトル（t_hat）
    polyvec_t u;                 // 暗号文の u 部分（多項体ベクトル）
    poly_t v;                    // 暗号文の v 部分（多項式）
```

```
    poly_t at_s_prime;              // A^T * s'
    poly_t e_prime;                 // エラー多項式 e'
    poly_t m;                       // メッセージ（多項式）
    poly_t pkv;                     // pk.t * v（検証用）

    uint8_t rho[KYBER_SYMMETRIC_BYTES];      // 公開鍵から取得する rho シード
    uint8_t mu[KYBER_SYMMETRIC_BYTES];       // メッセージシード
    uint8_t coin[KYBER_SYMMETRIC_BYTES];     // 乱数コイン
    uint8_t G_output[2 * KYBER_SYMMETRIC_BYTES]; // G 関数の出力（k || r）

    return 0; // 仮の実装
}

// 非カプセル化関数
// k: 共通鍵（バイト列）
// c: 暗号文（バイト列）
// sk: 秘密鍵（バイト列）
int crypto_kem_decapsulate_kyber512(uint8_t* k, const uint8_t* c, const uint8_t* sk) {
    polyvec_t sk_polyvec_decomp; // 展開された秘密鍵の多項式ベクトル（s）
    polyvec_t u_decomp;          // 展開された暗号文の u 部分
    poly_t v_decomp;             // 展開された暗号文の v 部分
    poly_t h_poly;               // s^T * u
    poly_t message_poly;         // 復元されたメッセージ多項式
    poly_t pkv_prime;            // pk.t * v_prime（再計算用）
```

```
    uint8_t tr_sk[KYBER_SYMMETRIC_BYTES]; //  秘密鍵から取得する tr シード
    uint8_t pki_sk[KYBER_SYMMETRIC_BYTES]; //  秘密鍵から取得する P_KI
    uint8_t mu_prime[KYBER_SYMMETRIC_BYTES]; //  復元されたメッセージシード
    uint8_t K_prime[KYBER_SYMMETRIC_BYTES];   //  復元された共通鍵  K'
    uint8_t d_prime[KYBER_SYMMETRIC_BYTES];   // Re-encryption  で使用されるハッシュ入力

    return 0; //  仮の実装
}
```

【#include "poly.h"】

```
#include "fips202.h" // SHAKE 関数を使用するため
#include "ntt.h"        // NTT 変換関数を使用するため
#include "kem.h"        // KYBER_Q, KYBER_N, KYBER_SYMMETRIC_BYTES  などのため（もし poly.h から直接参照できない場合も考慮）
#include <string.h>   // memcpy のため

//  多項式ベクトルの行列  A_hat  をシード  rho  から生成する関数
void gen_matrix(polyvec_t A[KYBER_K], const uint8_t rho[KYBER_SYMMETRIC_BYTES], int transpose) {
    unsigned int i, j;
```

```
    for (i = 0; i < KYBER_K; i++) {
        for (j = 0; j < KYBER_K; j++) {
            if (transpose) {
                // A[j].vec[i] に poly_uniform の結果を格納し、NTT ドメインに変換
                poly_uniform(&A[j].vec[i], rho, (i << 8) + j);
                poly_to_ntt(&A[j].vec[i]);
            }
            else {
                // A[i].vec[j] に poly_uniform の結果を格納し、NTT ドメインに変換
                poly_uniform(&A[i].vec[j], rho, (j << 8) + i);
                poly_to_ntt(&A[i].vec[j]);
            }
        }
    }
}

// poly_uniform: シードとノンスに基づいて、一様ランダムな多項式を生成します。
// Keccak からの出力を利用し、Rejection Sampling を行います。
// Kyber リファレンス実装の rej_uniform 関数に相当します。
void poly_uniform(poly_t* r, const uint8_t seed[KYBER_SYMMETRIC_BYTES], uint16_t
nonce) {
    unsigned int i = 0;
    unsigned int buflen = 0;
```

```
    unsigned int ctr = 0;
    uint16_t val;
    uint8_t buf[SHAKE128_RATE]; // SHAKE128 の内部レートと同じバッファサイズ
    uint8_t extseed[KYBER_SYMMETRIC_BYTES + 2]; //  シードとノンスを連結したバッファ

    // extseed  の構築: seed || nonce
    memcpy(extseed, seed, KYBER_SYMMETRIC_BYTES);
    extseed[KYBER_SYMMETRIC_BYTES] = nonce & 0xFF;
    extseed[KYBER_SYMMETRIC_BYTES + 1] = nonce >> 8;

    // SHAKE128 (XOF) を用いて乱数を生成し、Rejection Sampling  を行う
    shake128(buf, SHAKE128_RATE, extseed, sizeof(extseed)); //  最初のブロックを生成

    while (ctr < KYBER_N) {
        val = (uint16_t)buf[buflen] | ((uint16_t)buf[buflen + 1] << 8); // 2 バイトを読み込み

        if (val < KYBER_Q) { // Q=3329
            r->coeffs[ctr++] = (int16_t)val;
        }

        buflen += 2; // 2 バイト消費

        if (buflen + 2 > SHAKE128_RATE) { //  バッファが足りなくなったら新しいブロック
```

```
を生成
                // Kyber ref. では extseed の最終バイトをインクリメントし、SHAKE を再呼び出し
                // 実際は、nonce が次のブロックの生成に影響するようにする必要があります。
                // 簡略化のため、ここでは常に新しい乱数ブロックを同じ入力で生成するとします。
                // (Kyber リファレンスでは nonce をインクリメントして SHAKE を再利用し、新しい乱数を得る)
                // この部分は、より Kyber リファレンスの rej_uniform に合わせる必要があります。
                // ここでは簡略化のため、常に新しいブロックを再生成とします。
                shake128(buf, SHAKE128_RATE, extseed, sizeof(extseed)); // 同じ seed/nonce で呼び出すと常に同じシーケンス
                buflen = 0; // バッファをリセット
            }
        }

    // 係数を [-(Q-1)/2, (Q-1)/2] の範囲に正規化 (poly_uniform の最後に一度だけ行う Kyber ref.の挙動)
    for (i = 0; i < KYBER_N; i++) {
        if (r->coeffs[i] > (KYBER_Q - 1) / 2) {
            r->coeffs[i] -= KYBER_Q;
        }
    }
}
```

```
// 多項式をバイト列に圧縮する関数（実装）
void poly_compress(uint8_t* r, poly_t* a) {
    unsigned int i;
    int32_t val;

    for (i = 0; i < KYBER_N / 2; i++) { // 256/2 = 128 回ループ
        // 12 ビット圧縮（256 係数を 384 バイトに圧縮）
        // val は［-Q/2, Q/2］の範囲にある（約 -1664 to 1664）
        // Kyber ref: val = (((int32_t)a->coeffs[2*i] << 11) + KYBER_Q/2) / KYBER_Q;
        //                  r->coeffs[2*i]  の値を 0 から Q-1 の範囲に正規化してから圧縮するのが正しい
        // ここでは便宜上、直接ビット操作でパックします。
        // Kyber ref では、(val + KYBER_Q) % KYBER_Q のようにして正の範囲に正規化し、その後圧縮
        // 圧縮アルゴリズムは、上位 12 ビットと下位 4 ビットを組み合わせる
        val = a->coeffs[2 * i];
        r[3 * i + 0] = (uint8_t)(val & 0xFF);          // 下位 8 ビット
        r[3 * i + 1] = (uint8_t)(val >> 8);            // 残り 4 ビット
        val = a->coeffs[2 * i + 1];
        r[3 * i + 1] |= (uint8_t)(val << 4);           // 次の係数の下位 4 ビット
        r[3 * i + 2] = (uint8_t)(val >> 4);            // 次の係数の残り 8 ビット
    }
}
```

```
// バイト列から多項式を復元する関数（実装）
void poly_decompress(poly_t* r, const uint8_t* a) {
    unsigned int i;
    uint16_t val0, val1;

    for (i = 0; i < KYBER_N / 2; i++) {
        // 12 ビット伸長
        val0 = (uint16_t)a[3 * i + 0] | ((uint16_t)a[3 * i + 1] << 8);
        val1 = ((uint16_t)a[3 * i + 1] >> 4) | ((uint16_t)a[3 * i + 2] << 4);

        // 係数を [-(Q-1)/2, (Q-1)/2] の範囲に正規化
        r->coeffs[2 * i] = (int16_t)((val0 + KYBER_Q / 2) % KYBER_Q - KYBER_Q / 2);
        r->coeffs[2 * i + 1] = (int16_t)((val1 + KYBER_Q / 2) % KYBER_Q - KYBER_Q / 2);
    }
}

// 多項式を NTT ドメインに変換する関数（実装）
void poly_to_ntt(poly_t* r) {
    ntt(r); // ntt.c で定義した ntt 関数を呼び出す
}

// 多項式を通常のドメインに戻す関数（逆 NTT）（実装）
void poly_from_ntt(poly_t* r) {
    inv_ntt(r); // ntt.c で定義した inv_ntt 関数を呼び出す
```

```
}

// poly_zero: 多項式をゼロで初期化
void poly_zero(poly_t* r) {
    for (int i = 0; i < KYBER_N; i++) {
        r->coeffs[i] = 0;
    }
}

// polyvec_zero: 多項式ベクトルをゼロで初期化
void polyvec_zero(polyvec_t* r) {
    for (int i = 0; i < KYBER_K; i++) {
        poly_zero(&r->vec[i]);
    }
}
```

【include "ntt.h"】

```
#include "kem.h" // KYBER_N, KYBER_Q のため
#include <stddef.h> // size_t のため

#define Q KYBER_Q
#define QINV 62209 // inv(Q) mod 2^16 = -3327 (mod 2^16), Kyber のリファレンス実装に依存

// Kyber のリファレンス実装から取得した正確な zetas と zetas_inv 配列
```

// これは KYBER_N=256 の場合の定数です。

const int16_t zetas[128] = {

2285, 2568, 2981, 1056, 1729, 2925, 2351, 1827, 1928, 2872, 2197, 2697, 1374, 2580, 2908, 1146,

1216, 1867, 1289, 1779, 1926, 2769, 2439, 2095, 2696, 1381, 1559, 2384, 1852, 2118, 1422, 1902,

2874, 2859, 1378, 1489, 1509, 2361, 2360, 2736, 1695, 1754, 2146, 2891, 2946, 2883, 1144, 2221,

1312, 1686, 2665, 1746, 2735, 1461, 2174, 1858, 2235, 2172, 2404, 2320, 1900, 2852, 2038, 2577,

1802, 1519, 1838, 2796, 2963, 1073, 2906, 1709, 1702, 1916, 2085, 2561, 1152, 1618, 2307, 2707,

2115, 2642, 1693, 2140, 2201, 1599, 1362, 1970, 2575, 1982, 1683, 1937, 2663, 2607, 2259, 1475,

2205, 1269, 1789, 1403, 1346, 2029, 2626, 2007, 2671, 1972, 1122, 1150, 1326, 1383, 1690, 1464,

1442, 2617, 2471, 2730, 2542, 2017, 1731, 2520, 2771, 1588, 1163, 2596, 2496, 2167, 1243, 1421

};

const int16_t zetas_inv[128] = {

2409, 2377, 2345, 2313, 2281, 2249, 2217, 2185, 2153, 2121, 2089, 2057, 2025, 1993, 1961, 1929,

1897, 1865, 1833, 1801, 1769, 1737, 1705, 1673, 1641, 1609, 1577, 1545, 1513, 1481, 1449, 1417,

1385, 1353, 1321, 1289, 1257, 1225, 1193, 1161, 1129, 1097, 1065, 1033, 1001, 969, 937, 905,

873, 841, 809, 777, 745, 713, 681, 649, 617, 585, 553, 521, 489, 457, 425, 393,

361, 329, 297, 265, 233, 201, 169, 137, 105, 73, 41, 9, 3328, 3296, 3264, 3232,

3200, 3168, 3136, 3104, 3072, 3040, 3008, 2976, 2944, 2912, 2880, 2848, 2816, 2784, 2752, 2720,

2688, 2656, 2624, 2592, 2560, 2528, 2496, 2464, 2432, 2400, 2368, 2336, 2304, 2272, 2240, 2208,

2176, 2144, 2112, 2080, 2048, 2016, 1984, 1952, 1920, 1888, 1856, 1824, 1792, 1760, 1728, 1696

};

```
// montgomery_reduce: x * 2^-16 mod Q を計算（正確な実装）
int16_t montgomery_reduce(int32_t a) {
    int32_t t;
    int16_t u;

    u = (int16_t)(((uint16_t)a * QINV)); // QINV は -Q^(-1) mod 2^16
    t = (int32_t)u * Q;
    t = a - t;
    t >>= 16; // 2^16 で割る

    return t;
}
```

```
// basemul: NTT ドメインでの多項式乗算（a[0]*b[0] + a[1]*b[1] mod (X^2-zeta)）
// r: 結果の多項式
// a: 入力多項式 a
// b: 入力多項式 b
static void basemul(int16_t* r, const int16_t* a, const int16_t* b, int16_t zeta) {
    int32_t t;
    t = (int32_t)a[0] * b[0];
    t = montgomery_reduce(t); // a0*b0
    r[0] = (int16_t)t;

    t = (int32_t)a[1] * b[1];
    t = montgomery_reduce(t); // a1*b1
    t = (int32_t)zeta * t;
    t = montgomery_reduce(t); // zeta * a1*b1

    r[0] = (int16_t)(r[0] + t); // r0 = a0*b0 + zeta*a1*b1

    t = (int32_t)a[0] * b[1];
    t = montgomery_reduce(t); // a0*b1
    r[1] = (int16_t)t;

    t = (int32_t)a[1] * b[0];
    t = montgomery_reduce(t); // a1*b0
    r[1] = (int16_t)(r[1] + t); // r1 = a0*b1 + a1*b0
```

```
}

// NTT (Number Theoretic Transform)
void ntt(poly_t* r) {
    int i, j, k;
    int16_t* coeffs = r->coeffs;
    int16_t zeta;
    int len, start;

    len = 128; // KYBER_N/2

    // 最初の層のバタフライ演算
    for (i = 0; i < len; i++) {
        zeta = zetas[i];
        for (j = i; j < KYBER_N; j += 2 * len) {
            int16_t* a = &coeffs[j];
            int16_t* b = &coeffs[j + len];

            int32_t t = (int32_t)*b * zeta;
            t = montgomery_reduce(t);

            *b = (int16_t)(*a - t);
            if (*b < 0) *b += Q;
            *a = (int16_t)(*a + t);
```

```
            if (*a >= Q) *a -= Q;
        }
    }

    len = 64; // 次の層
    start = 0; // zetas_inv の開始インデックス

    // 残りの層
    for (k = 1; k < 8; k++) { // log2(N) - 1 層（N=256 なので 8 層）
        for (i = 0; i < len; i++) {
            zeta = zetas[start + i]; // zetas_inv ではなく zetas を使用（NTT の場合）
            for (j = i; j < KYBER_N; j += 2 * len) {
                int16_t* a = &coeffs[j];
                int16_t* b = &coeffs[j + len];

                int32_t t = (int32_t)*b * zeta;
                t = montgomery_reduce(t);

                *b = (int16_t)(*a - t);
                if (*b < 0) *b += Q;
                *a = (int16_t)(*a + t);
                if (*a >= Q) *a -= Q;
            }
        }
```

```
            len /= 2;
            start += 2 * len; // zetas 配列のインデックスの更新
        }
    }

// Inverse NTT
void inv_ntt(poly_t* r) {
    int i, j, k;
    int16_t* coeffs = r->coeffs;
    int16_t zeta;
    int len, start;

    len = 1; // 逆 NTT の最初の層

    // 逆 NTT の層
    for (k = 0; k < 8; k++) { // log2(N) 層 (N=256 なので 8 層)
        start = 128 - (len * 2); // zetas_inv の開始インデックス
        for (i = 0; i < len; i++) {
            zeta = zetas_inv[start + i];
            for (j = i; j < KYBER_N; j += 2 * len) {
                int16_t* a = &coeffs[j];
                int16_t* b = &coeffs[j + len];

                int32_t t_plus = (int32_t)*a + *b;
```

```
                int32_t t_minus = (int32_t)*a - *b;
                if (t_minus < 0) t_minus += Q;

                *a = (int16_t)t_plus;
                if (*a >= Q) *a -= Q;

                int32_t t = (int32_t)t_minus * zeta;
                *b = montgomery_reduce(t);
            }
        }
        len *= 2;
    }
    // 最終的なスケール調整（N^-1 mod Q = 1/256 mod 3329）
    // 256^-1 mod 3329 は 3329 * x + 256 * y = 1 から計算
    // 256 * 2995 = 766720
    // 766720 mod 3329 = 1
    // つまり、N_inv = 2995 です。
    const int16_t N_inv = 2995; // 256^-1 mod 3329

    for (i = 0; i < KYBER_N; i++) {
        r->coeffs[i] = montgomery_reduce((int32_t)r->coeffs[i] * N_inv);
    }
}
// poly_basemul_montgomery: NTT ドメインでの多項式乗算（r = a * b）
```

```
void poly_basemul_montgomery(poly_t* r, const poly_t* a, const poly_t* b) {
    for (int i = 0; i < KYBER_N / 2; i++) {
        basemul(&r->coeffs[2 * i], &a->coeffs[2 * i], &b->coeffs[2 * i], zetas[64 + i]); // zeta は zetas の後半を使用
    }
}
```

【#include "fips202.h"】

```
#include <string.h>

typedef uint64_t state_t[25];

#define ROL64(a, offset) (((a) << (offset)) ^ ((a) >> (64 - (offset))))

const uint64_t KeccakF_RC[] = {
    0x0000000000000001ULL, 0x0000000000008082ULL, 0x800000000000808aULL,
    0x8000000080008000ULL, 0x0000000000008081ULL, 0x0000000000008006ULL,
    0x0000000080000003ULL, 0x8000000080000002ULL, 0x0000000000000080ULL,
    0x000000000000800aULL, 0x000000008000000aULL, 0x0000000080008081ULL,
    0x0000000000008080ULL, 0x8000000000000001ULL, 0x8000000080008081ULL,
    0x8000000080008006ULL, 0x8000000080000003ULL, 0x8000000080000002ULL,
    0x8000000000000080ULL, 0x000000000000800aULL, 0x800000008000000aULL,
    0x8000000080008081ULL, 0x8000000000008080ULL, 0x0000000000000001ULL
```

```
};

static void KeccakF1600_StatePermute(state_t state) {
    int i, j, round;
    uint64_t C[5], D[5];
    uint64_t current;

    const int KeccakRhoOffsets[25] = {
         0, 1, 62, 28, 27,
        36, 44, 6, 55, 20,
         3, 10, 43, 25, 39,
        41, 45, 15, 21, 8,
        18, 2, 61, 56, 14
    };

    const int KeccakPiOffsets[25] = {
         0, 1, 5, 2, 8,
         9, 13, 16, 3, 17,
        18, 20, 21, 10, 22,
        23, 6, 7, 11, 24,
        12, 14, 15, 4, 19
    };

    for (round = 0; round < 24; round++) {
```

```
for (i = 0; i < 5; i++) {
    C[i] = state[i] ^ state[i + 5] ^ state[i + 10] ^ state[i + 15] ^ state[i + 20];
}
for (i = 0; i < 5; i++) {
    D[i] = C[(i + 4) % 5] ^ ROL64(C[(i + 1) % 5], 1);
}
for (i = 0; i < 25; i++) {
    state[i] ^= D[i % 5];
}

current = state[1];
for (i = 0; i < 24; i++) {
    j = KeccakPiOffsets[i + 1];
    uint64_t temp = state[j];
    state[j] = ROL64(current, KeccakRhoOffsets[j]);
    current = temp;
}

for (j = 0; j < 25; j += 5) {
    for (i = 0; i < 5; i++) {
        C[i] = state[j + i];
    }
    for (i = 0; i < 5; i++) {
        state[j + i] = C[i] ^ ((~C[(i + 1) % 5]) & C[(i + 2) % 5]);
```

```
                }
            }
            state[0] ^= KeccakF_RC[round];
        }
    }

static void shake_init(state_t state) {
    memset(state, 0, sizeof(state_t));
}
static void shake_absorb(state_t state, const uint8_t* in, size_t inlen, size_t rate) {
    size_t i;
    for (i = 0; i < inlen / rate; i++) {
        for (size_t k = 0; k < rate / 8; ++k) {
            state[k] ^= ((uint64_t*)in)[i * (rate / 8) + k];
        }
        KeccakF1600_StatePermute(state);
    }
    size_t remaining_bytes = inlen % rate;
    if (remaining_bytes > 0) {
        uint8_t partial_block[SHAKE256_RATE];
        memcpy(partial_block, in + i * rate, remaining_bytes);
        memset(partial_block + remaining_bytes, 0, rate - remaining_bytes);

        for (size_t k = 0; k < rate / 8; ++k) {
```

```
                state[k] ^= ((uint64_t*)partial_block)[k];
            }
        }
    }

    static void shake_squeeze(state_t state, uint8_t* out, size_t outlen, size_t rate) {
        size_t i;
        for (i = 0; i < outlen / rate; i++) {
            memcpy(out + i * rate, (uint8_t*)state, rate);
            KeccakF1600_StatePermute(state);
        }
        size_t remaining_bytes = outlen % rate;
        if (remaining_bytes > 0) {
            memcpy(out + i * rate, (uint8_t*)state, remaining_bytes);
        }
    }

    void shake128(uint8_t* output, size_t outlen, const uint8_t* input, size_t inlen) {
        state_t state;
        shake_init(state);
        shake_absorb(state, input, inlen, SHAKE128_RATE);

        state[SHAKE128_RATE / 8 - 1] ^= 0x80ULL << 56;
        state[SHAKE128_RATE / 8 - 1] ^= 0x01ULL << 63;
        KeccakF1600_StatePermute(state);
        shake_squeeze(state, output, outlen, SHAKE128_RATE);
```

```
}
void shake256(uint8_t* output, size_t outlen, const uint8_t* input, size_t inlen) {
    state_t state;
    shake_init(state);
    shake_absorb(state, input, inlen, SHAKE256_RATE);

    state[SHAKE256_RATE / 8 - 1] ^= 0x80ULL << 56;
    state[SHAKE256_RATE / 8 - 1] ^= 0x01ULL << 63;

    KeccakF1600_StatePermute(state);
    shake_squeeze(state, output, outlen, SHAKE256_RATE);
void sha3_256(uint8_t* output, const uint8_t* input, size_t inlen) {
    // 実装は後ほど追加します
}
void sha3_512(uint8_t* output, const uint8_t* input, size_t inlen) {
    // 実装は後ほど追加します
}
```