明治大学理工学部情報科学科

ソフトウェア実習 レポート

6章 スタック

提出日 2025 年 11 月 17 日

3 年 15組 91番 157R257501

SEO BEOMGYO

はじめに

本課題の目的は，スタックを用いた後置記法電卓を実装する過程を通じて，式の構文解析と実行時エラー処理の基礎を理解し，整数演算における境界条件やオーバーフローの扱いを学ぶことである。
特に，配列によるスタック実装，トークン列からの演算手続の構成，ユーザにとって分かりやすいエラーメッセージ設計，さらに32bit int 型の表現範囲を前提としたオーバーフロー検出を重点的に確認した。

課題1

```c
void initialize_stack(void)
{
    sp = -1;
}
int pop(void)
{
    if (sp >= 0)
        return stack[sp--];   // 値をポップして返す
    else {
        printf("stack empty error.\n");
        exit(1);
    }
}
int main(void)
{
    int x;
    c = getchar();
    token = get_token();
    initialize_stack();
    while (token != END) {
        switch (token) {
            case NUMBER: push(num); break;
            case PLUS:   push(pop() + pop()); break;
            case MINUS:  x = pop(); push(pop() - x); break;
            case MULT:   push(pop() * pop()); break;
            case DIV:    x = pop(); push(pop() / x); break;
            default:     printf("illegal character.\n"); exit(1);
        }
        token = get_token();
    }
    if (sp == 0) printf("%10d\n", pop());
    else { printf("expression syntax error.\n"); exit(1); }
    return 0;
}
```

- 目的：スタックを用いて，加減乗除を含む後置記法式を評価する基本的な電卓プログラムを完成させる。
- 配列 stack[STACK_SIZE] とスタックポインタ sp を用い，initialize_stack，pus

h, pop を実装してスタック操作を提供した。

- get_token 関数で空白類を読み飛ばしつつ整数と演算子を識別し，main ではトークン列を読みながら演算子ごとに pop した 2 つのオペランドに対して演算を行い，結果を再び push することで式全体を評価した。
- テストと実行結果
    - 入力例として，テキストの例に従い 3 4 + 5 * と 3 4 5 * + を与えた
    - 出力はそれぞれ 35，23 となり，手計算の結果と一致することを確認した。

課題3

```
typedef enum { NUMBER, PLUS, MINUS, MULT, DIV, MOD, POW, OTHER, END } ttyp;
ttyp token;

int c, num;

static int ipow(int base, int exp)
{
    // 整数べき乗（指数は0以上）
    if (exp < 0) { printf("negative exponent.\n"); exit(1); }
    int result = 1;
    while (exp) {
        if (exp & 1) result *= base;
        base *= base;
        exp >>= 1;
    }
    return result;
}
```

```
case '%': c = getchar(); return MOD;
case '^': c = getchar(); return POW;
```

```
case MOD:    x = pop(); push(pop() % x); break;
case POW:    x = pop(); push(ipow(pop(), x)); break;
```

- 目的：基本電卓に新たな2項演算子としてべき乗演算と剰余演算 % を追加し，後置記法でより複雑な整数式を扱えるようにする。
- 実装要点：列挙型 ttyp に MOD，POW を追加し，get_token で記号 '%'，'^' をそれぞれのトークン種別に対応させた。
  整数べき乗は補助関数 ipow(base, exp) を定義し，指数が 0 以上であることを前提に二乗反復法を用いて O(log exp) 回の乗算で計算するようにした。
  メインループでは MOD の場合に x = pop(); push(pop() % x);，POW の場合に x = pop(); push(ipow(pop(), x)); とし，MINUS/DIV と同様にオペランドの順序を保つよう注意した。
- テストと実行結果
    - 課題文の例に従い，2 5 ^ と 5 2% を入力した。
      出力はそれぞれ 32 と 1 となり，$2^5=32$ および 5 ÷ 2 の余りが 1 であることから，実装が正しいことを確認した。
      さらに 2 5 ^ 5 2% + のように複数の演算子を組み合わせた式についても手計算と一致する結果が得られた。

課題5

```c
void push(int x)
{
    if (sp < STACK_SIZE - 1)
        stack[++sp] = x;
    else
        user_error("式が長すぎます（スタックが満杯です）");
}

int pop(void)
{
    if (sp >= 0)
        return stack[sp--];
    else
        user_error("オペランドが不足しています"); // スタック空
    return 0;
}

// 整数べき乗（指数は0以上）
static int ipow(int base, int exp)
{
    if (exp < 0) user_error("負の指数は扱えません");
    int result = 1;
    while (exp) {
        if (exp & 1) result *= base;
        base *= base;
        exp >>= 1;
    }
    return result;
}
```

- 目的：誤った入力データに対して，内部仕様依存のメッセージではなく，ユーザの視点からできるだけ分かりやすい日本語のエラーメッセージを出力するようプログラムを改善する。
- 実装要点：user_error 関数を導入し，すべての致命的エラーをこの関数経由で処理することでメッセージ形式と終了処理を一元化した。
  peek_next_is_number と require_two を用意し，2項演算子の直前にスタック上のオペランド数を検査するとともに，sp == 0 かつ次のトークンが数値である場合には 1 + 2 のような中置記法と判断して「演算子に対応するオペランドの順序が正しくありません（後置記法で入力してください：例 1 2 +）」と具体的に指摘するようにした。
  スタックあふれ，空スタックからの pop，0 での除算，不正文字の検出など，それぞれの状況に応じて「式が長すぎます（スタックが満杯です）」「オペランドが不足しています」「0 で割ることはできません」「無効な文字です：'x'」といった日本語メッセージを出すよう修正した。
- テストと実行結果
  - 正しい入力として 3 4 + 5 * や 2 5 ^ 5 2% + を与えた場合には，これまで通り正しい数値が出力され，エラーメッセージは発生しないことを確認した。
    誤った例として 1 + 2，12 + *，1 2 + 0，5 0 /，および英字を含む式を与えたところ，それぞれ順序エラー，オペランド不足，オペレータ不足，0 除算，不正トークンとして意図した日本語メッセージが表示された。

課題6

```
static void eval_line(char* line) {
    init_stack();
    char* p = line;
    int had_input = 0;

    while (*p) {
        while (isspace((unsigned char)*p)) p++;
        if (!*p) break;

        if (*p == '+' || *p == '-' || *p == '*' || *p == '/' || *p == '%' || *p == '^') {
            char op = *p++;
            if (op == '+') {
                if (sp < 1) user_error("オペランドが不足しています (+)");
                push(pop() + pop());
            } else if (op == '-') {
                if (sp < 1) user_error("オペランドが不足しています (-)");
                int b = pop(), a = pop();
                push(a - b);
            } else if (op == '*') {
                if (sp < 1) user_error("オペランドが不足しています (*)");
                push(pop() * pop());
            } else if (op == '/') {
                if (sp < 1) user_error("オペランドが不足しています (/)");
                int b = pop(); if (b == 0) user_error("0 で割ることはできません");
                int a = pop();
                push(a / b);
            } else if (op == '%') {
                if (sp < 1) user_error("オペランドが不足しています (%)");
                int b = pop(); if (b == 0) user_error("0 で割ることはできません");
                int a = pop();
                push(a % b);
            } else {
                if (sp < 1) user_error("オペランドが不足しています (^)");
                int e = pop(), a = pop();
                push(ipow(a, e));
            }
            had_input = 1;
            continue;
        }

        char* endp;
        long v = strtol(p, &endp, 10);
```

- 目的: 1 行ごとに後置記法式を読み取り，行末をもってその式の計算を確定し，結果を出力したのち次の行の入力を受け付けるインタラクティブな電卓に拡張する。
- 実装要点: 標準入力から fgets で 1 行ずつ読み取り，各行に対して eval_line 関数を呼び出してスタックを初期化した上でトークン解析と評価を行う構成とした。行内の解析では isspace と strtol を用いて数値と演算子を切り出し，設定したスタックあふれ・オペランド不足・0 除算・不正トークンなどの検査と日本語エラーメッセージ出力を再利用した。
  整数の入力値については long で一旦受け取り，INT_MIN～INT_MAX の範囲外であれば「整数の範囲を超えています」として弾くことで，入力段階での桁あふれも防いだ。
- テストと実行結果
  - ターミナル上でプログラムを起動し，1 行目に 1 2 +，2 行目に 2 3 * と入力したところ，それぞれの直後に 3，6 が別行で表示され，引き続き次の式の入力を受け付けることを確認した。
    また空行や空白のみの行に対しては計算を行わずスキップされること，範囲外の大きな整数や不正な文字列を含む行では，行単位で日本語エラーメッセージが表示されることを確認した。

課題7

```c
static int mul_checked(int a, int b, int* out) {
    if (a == 0 || b == 0) { *out = 0; return 0; }
    if (a == -1 && b == INT_MIN) return 1;
    if (b == -1 && a == INT_MIN) return 1;
    if (a > 0) {
        if (b > 0) { if (a > INT_MAX / b) return 1; }
        else       { if (b < INT_MIN / a) return 1; }
    } else {
        if (b > 0) { if (a < INT_MIN / b) return 1; }
        else       { if (a < INT_MAX / b) return 1; }
    }
    *out = a * b; return 0;
}
```

- 目的：32bit の int 型で表現できる範囲 [−2 147 483 648,2 147 483 647] を超える演算結果を検出し，オーバーフロー発生時に適切な日本語メッセージを出力する機能を追加する。
- 実装要点：<limits。h> をインクルードし，INT_MAX，INT_MIN を用いて加算，減算，乗算，除算，剰余，べき乗それぞれに対して事前または逐次的な範囲チェックを行う補助関数群 add_checked，sub_checked，mul_checked，div_checked，mod_checked，ipow_checked を実装した。
  加算と減算では，被演算数と加減数の符号に応じて a > INT_MAX - b や a < INT_MIN + b といった条件でオーバーフローを判定し，乗算では符号の組み合わせごとに INT_MAX / b，INT_MIN / b との大小比較で安全性を判断する方法を用いた。
  除算に関しては，0 除算を禁止するとともに，唯一のオーバーフローケースである INT_MIN / -1 を特別扱いし，剰余やべき乗ではそれぞれ mod_checked，ipow_checked の内部で mul_checked を繰り返し利用することで中間結果のあふれも検出した。
  演算結果が範囲外と判定された場合には，「演算結果が範囲を超えました（加算オーバーフロー）」など演算種別を含む日本語メッセージを出力して即時にプログラムを終了するようにした。
- テストと実行結果
  - 加算の例として 2147483647 1 + を入力したところ，「演算結果が範囲を超えました（加算オーバーフロー）」というメッセージが表示され，異常値が出力されないことを確認した。
    乗算の例として 50000 50000 * を与えた場合にも，「演算結果が範囲を超えました（乗算オーバーフロー）」が表示され，内部での事前チェックが有効に働いていることを確認した。
    除算では -2147483648 -1 / に対し「演算結果が範囲を超えました（除算オーバーフロー）」が出力される一方，0 除算 5 0 / では従来通り「0 で割ることはできません」が表示され，それぞれの異常が区別されて扱われることを確認した。

考察

最後に，int 型の表現範囲や未定義動作の扱いなど，言語仕様レベルの制約を意識してプログラムを設計することの重要性を再認識した。