プログラミング通論'20 ＃3 – スタックとその利用

久野 靖 (電気通信大学)

2020.2.21

今回は次のことが目標となります。

- CSにおける基本的なデータ構造であるスタックの機能と実装について知る。
- スタックを用いた基本的なアルゴリズムについて知る。

その前にコマンド引数の話題から始め、以後プログラムへの入力にコマンド引数を併用します。

コマンド引数

コマンド引数 (comand line arguments) とはもともとUnixの用語で、コマンドやプログラムを起動するときに「`./a.out aa bbb ccc`」のようにコマンド名(プログラム名)より後ろに指定する文字列のことです。C言語では`main`の引数を`void`と指定するかわりに、次のように指定することでこの情報を受け取れます。

```
int main(int argc, char *argv[]) {
  ...
```

コマンド引数

argc、argv は引数なので名前は任意ですが、これらの名前を使うことが慣例となっています。これらに渡される内容ですが、argc はコマンド引数の語の数 (コマンド名を含む)、argv はその各要素がそれぞれの語の文字列 (C 言語なので char へのポインタ) となっています (図 1)。

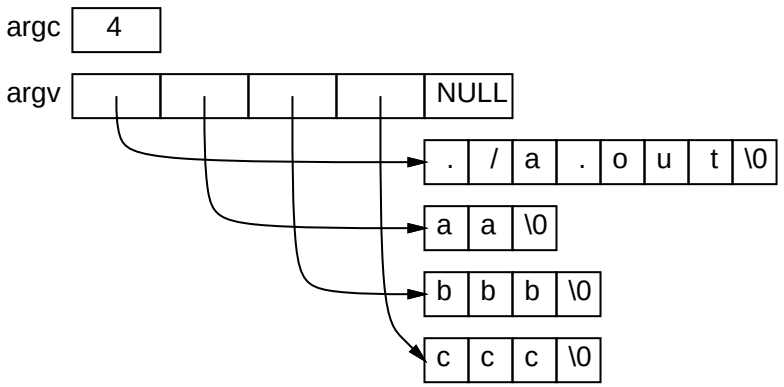

図 1: コマンド引数の構造

コマンド引数

ではこれらを文字列として打ち出して見ましょう。

```
// argdemo1.c --- print argc/argv as string
#include <stdio.h>
int main(int argc, char *argv[]) {
  for(int i = 0; i < argc; ++i) { printf("%s\n", argv[i]); }
  return 0;
}
```

実行のようすは次の通り。「0番目」はコマンド名になること、「'...'」や「"..."」で囲んだものは1つの文字列として渡されることに注意(Cでは文字列は「"..."」のみですが、シェルやRubyでは「'...'」でもよいのでしたね)。

```
% ./a.out aa 'bbb ccc'
./a.out
aa
bbb ccc
```

コマンド引数

ついでにもう1つ、数値を受け取りたい場合には文字列から整数や実数に変換する次の関数が使えます(stdlib.hで定義されています)。

- `int atoi(char *s)` — 文字列をそれが表す整数に変換。
- `double atof(char *s)` — 文字列をそれが表す実数値に変換。

これも例を示しましょう。コマンド名は数でないので、今度は1番目から順に処理します。

コマンド引数

```
// argdemo2.c --- sum of argument strings (as int)
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
  int sum = 0;
  for(int i = 1; i < argc; ++i) { sum += atof(argv[i]); }
  printf("sum = %d\n", sum);
  return 0;
}

% ./a.out 3 4 5
sum = 12
```

この先ではscanfによる入力に加えて、このコマンド行引数による入力も活用していきます。

スタック

スタックの概念

スタック (stack) とは「積み上げたもの」を意味しますが、コンピュータサイエンスでは**LIFO**(last-in, first-out) すなわち「後に入れたものほど先に出て来る」記憶領域のことを指します。図2左のように、「1」「2」「3」がこの順でやってきたとき、記憶領域に「積み上げて」保管すると、取り出した時に逆順になっていますよね。つまり「後から入れたものほど先に出て来る」からそうなるわけです。

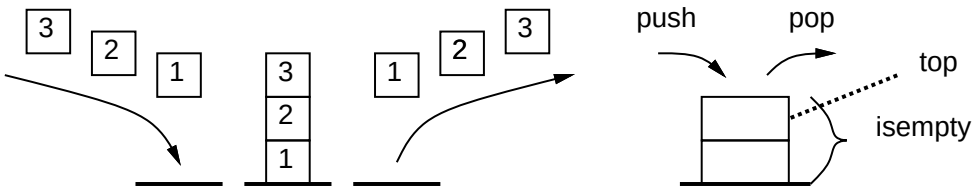

図 2: スタックの概念

スタックをプログラム上で実現する場合、その基本操作は要素を追加する**push**、そして取り出す**pop**です (日本語ではしばしば「積む」「降ろす」とも言います)。また、空っぽになったのにさらに取り出すわけには行きませんから、空っぽかどうかを調べる操作 `isempty` が必要です。

スタックの概念

あと、先頭 (いちばん上) にある要素を降ろさずに見たいことがあるので、そのための操作 top を用意することもあります (push と pop があれば作れますが)。

ではスタックは何の役に立つのでしょう。それをこれから色々見ていきますが、まずは上の例のように「列を逆順にする」のに使えます。そんな面倒なことをしなくても、配列に入れて逆から取り出せばよい、とか思いましたか? そうなのですが、そのコードは結構面倒ですよね。それと比較して、「順に push して、それから順に pop する」方が簡単なのです。大した簡単さではないと思うかも知れませんが、このような単純化 (抽象化) が役に立つことは多くあるのです。

配列を使ったスタックの実装

スタックを実装する1つの方法は、配列を使うことです。配列arr と、次に入れる要素の位置を示す整数ptrを用いた場合、pushとpopは次のように書けます(ptrの初期値は0にします)。

```
arr[ptr++] = value;  // same as: arr[ptr] = value; ++ptr;
value = arr[--ptr];  // same as: --ptr; value = arr[ptr];
```

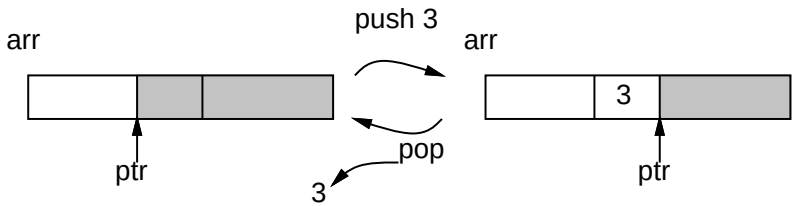

図 3: 配列を使ったスタックの実装

つまり、pushは「空いている位置に値を入れ、位置は1つ進める」、popは「位置を1つ戻し、その位置の値を取り出す」なわけです(図3)。ちょっと違う版として、ptrは空いている位置ではなく「最後に入れた値の位置」にする方法もありますが、その場合ptrの初期値は-1にする必要があります。

配列を使ったスタックの実装

では次に、これを前回やったレコード型を使ってカプセル化しましょう。さらに、上ではpushとpopだけでしたが、そのほかの機能も追加します。まずヘッダファイルから。

```
// istack.h --- int type stack interface
#include <stdbool.h>
struct istack;
typedef struct istack *istackp;
istackp istack_new(int size); // allocate new stack
bool istack_isempty(istackp p); // test if the stack is empty
void istack_push(istackp p, int v); // push a value
int istack_pop(istackp p);      // pop a value and return it
int istack_top(istackp p);      // peek the topmost value
```

配列を使ったスタックの実装

typedefという見慣れない行がありますが、この行はistackpという名前を「struct istack*」という型を表す型名として定義しています。以後、この名前を使うことで記述が簡潔になります。[1]

そして実装本体は次のようになります。作成時には十分余裕のある要素数を渡すものとします。先の説明で単独変数のように書いていたものはすべて、構造体のフィールドになり、アロー記法でアクセスします。topはptrの指す1つ手前の位置になることに注意。

[1]たとえば「typedef int *intp」により intp という名前を整数へのポインタを表す型として定義することもできますが、大して読みやすくはならないので、こちらはあまりやりません。

配列を使ったスタックの実装

```
// istack.c --- int type stack impl. with array
#include <stdlib.h>
#include "istack.h"
struct istack { int ptr; int *arr; };
istackp istack_new(int size) {
  istackp p = (istackp)malloc(sizeof(struct istack));
  p->ptr = 0; p->arr = (int*)malloc(size * sizeof(int));
  return p;
}
bool istack_isempty(istackp p) { return p->ptr <= 0; }
void istack_push(istackp p, int v) { p->arr[p->ptr++] = v; }
int istack_pop(istackp p) { return p->arr[--(p->ptr)]; }
int istack_top(istackp p) { return p->arr[p->ptr - 1]; }
```

配列を使ったスタックの実装

ではこれを使ってみることにして、文字列をさかさまにして出力するという例題を作ってみましょう。文字列はコマンド行から与えます。

```
// reversestr.c --- print argv[1] in reverse order
#include <stdio.h>
#include <stdlib.h>
#include "istack.h"
int main(int argc, char *argv[]) {
  istackp s = istack_new(200);
  char *t = argv[1];
  for(int i = 0; t[i] != '\0'; ++i) { istack_push(s, t[i]); }
  while(!istack_isempty(s)) { putchar(istack_pop(s)); }
  putchar('\n');
  return 0;
}
```

配列を使ったスタックの実装

呼び方は「./a.out ’文字列’」のようにするので、文字列はargv[1]で渡されますが、それを変数tに入れます。そしてその各文字をスタックに積んで行きます(文字列はナル文字’\0’で終わりになることに注意)。積み終ったらスタックから各文字を降ろして出力します。putchar(文字)は文字を1文字出力する関数です。実行例は次の通り。

```
% ./a.out 'this is a pen.'
.nep a si siht
```

配列を使ったスタックの実装

スタックの単体テストをしておきましょう。取り出した整数が正しいかチェックするので、expect_int を使っています。

```
// test_istack.c --- unit test for istack.
#include <stdio.h>
#include <stdlib.h>
#include "istack.h"
void expect_int(int i1, int i2, char *msg) {
  printf("%s %d:%d %s\n", (i1==i2)?"OK":"NG", i1, i2, msg);
}
int main(int argc, char *argv[]) {
  struct istack *s = istack_new(200);
  istack_push(s, 1); istack_push(s, 2); istack_push(s, 3);
  expect_int(istack_pop(s), 3, "push 1 2 3 ; pop");
  expect_int(istack_pop(s), 2, "pop");
  istack_push(s, 4);
  expect_int(istack_pop(s), 4, "push 4; pop");
```

```
  expect_int(istack_pop(s), 1, "pop");
  return 0;
}
```

実行例は次の通り。

```
% ./a.out
OK 3:3 push 1 2 3 ; pop
OK 2:2 pop
OK 4:4 push 4; pop
OK 1:1 pop
```

数式の扱い
括弧の対応

それではいよいよ、スタックが役に立つ例を見ていただくことにします。スタックは基本的に「入れ子構造」の処理に適しています。たとえば、数式などで括弧を使う場合、(1) 開き括弧と閉じ括弧が対応していて、(2) 閉じ括弧が先行することは無く、(3) 開きと閉じの対応関係がクロスしない、という条件が必要です (図 4)。これをチェックするアルゴリズムを考えてみましょう。

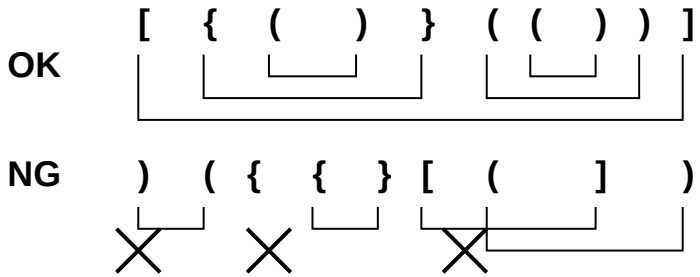

図 4: 括弧入れ子構造

括弧の対応

とりあえず簡単のため、丸括弧だけで考えます。図5では、左側にスタックを横向き(右に伸びる)に描き、右側に入力文字列を描いています(最後の$は入力終わりの印のつもり)。↑は入力の位置です。

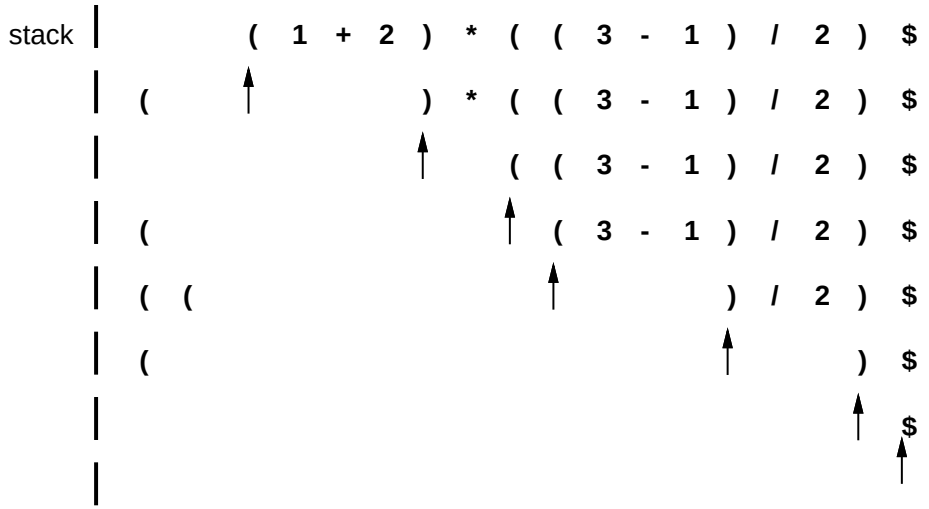

図 5: スタックを用いた括弧の対応チェック

括弧の対応

開き括弧が来たら、それをスタックに積みます。閉じ括弧が来たら、スタックに対応する括弧が載っていることを確認して、それを降ろします。問題なく最後まで来た時にちょうどスタックが空なら、対応が取れていたと分かります。途中で空のスタックから降ろそうとしたり、閉じ括弧とスタック上の開き括弧の種類が違っていたら、間違いがあったと分かります (ここでは丸括弧しか出ていませんが)。

では上に説明したアルゴリズムを C プログラムにしたものを示します。文字列を受け取って対応を調べ成否を返す関数が `balance1` です。その中でスタックを作り、文字列の各文字をループで順番に変数 `c` に取り出します。その文字が「`'('`」ならスタックに積み、「`')'`」であればスタックから降ろします。積んでいるのは「`'('`」だけなので、降ろすときにその文字が何であるかチェックする必要はありません。降ろそうとして空なら、対応があっていない (「`')'`」が余分) なので「いいえ」を返します。文字列の最後まで来たときはスタックが空なら対応が合っている (余分な「`'('`」が無い) ので、「空か否か」の論理値を返します。

`main` はコマンド引数の文字列 1 つずつを `balance1` に渡し、結果を文字列「OK」「NG」で表示するだけです。

括弧の対応

```
// barance1.c --- see if parentheses are balanced in input
#include <stdio.h>
#include <stdlib.h>
#include "istack.h"
bool balance1(char *t) {
  istackp s = istack_new(200);
  for(int i = 0; t[i] != '\0'; ++i) {
    char c = t[i];
    if(c == '(') {
      istack_push(s, c);
    } else if(c == ')') {
      if(istack_isempty(s)) { return false; }
      istack_pop(s);
    }
  }
  return istack_isempty(s);
```

```
}
int main(int argc, char *argv[]) {
  for(int i = 1; i < argc; ++i) {
    printf("%s : %s\n", argv[i], balance1(argv[i])?"OK":"NG");
  }
  return 0;
}
```

実行例を示します。

```
% ./a.out '((a))' '(a))'
((a)) : OK
(a)) : NG
```

括弧の対応

これも単体テストを作成してみました。bool2str、expect_bool は前回と同じです。

```
// test_barance1.c --- unit test for balance1
#include <stdio.h>
#include <stdlib.h>
#include "istack.h"
char *bool2str(bool b) { return b ? "true" : "false"; }
void expect_bool(bool b1, bool b2, char *msg) {
  printf("%s %s:%s %s\n", (b1==b2)?"OK":"NG",
         bool2str(b1), bool2str(b2), msg);
}
(balance1 here)
int main(void) {
  expect_bool(balance1("((a)())"), true, "((a)())");
  expect_bool(balance1(")(a)()("), false, ")(a)()(");
  expect_bool(balance1("((a)()"), false, "((a)()");
```

```
  expect_bool(balance1("(a())"), false, "(a())");
  expect_bool(balance1("((()))"), true, "((()))");
  return 0;
}
```

実行例は次の通り。

```
% ./a.out
OK true:true ((a))
OK false:false )(a)()(
OK false:false ((a)()
OK false:false (a))
OK true:true ((()))
```

括弧の対応

演習 1 括弧の対応プログラムをそのまま動かせ。正しい入力、正しくない入力、不自然だけど正しい入力などをそれぞれ複数試してみること。その後、次の課題をやってみよ。必ず単体テストを作成すること。

a. 括弧の種類として「`()`」に加え、「`[]`」と「`{}`」を扱えるようにしてみよ。ただし、どの開きかっことどの閉じかっこでも対応できるとしてよい (たとえば「`[(]}`」でも OK とする)。

b. 上と同じだが、同じ種類の開きかっこと閉じかっこが対応していなければいけないようにする (たとえば「`[(]}`」はだめで「`[()]`」は OK とする)。

c. 上と似ているが、それぞれのかっこは独立に対応をチェックする (かっこ同士が互い違いでもよい) ようにする (たとえば「`[(])`」は OK で「`[{)]`」はだめとする)。
ヒント: スタックを 3 本使う。

括弧の対応

d. 上のどれでかで間違いが発見された際、その位置が分かるようにしてみよ。(ヒント: 簡単には間違い発見時にその位置が何文字目か表示したり、次の行にその文字数-1個の空白の後印を出力する。ただしこの方法だと、括弧の対応が合わないときに該当の開き括弧がどれかは示せない。対応策としては、スタックをもう1つ用意して、開き括弧を積むときに何文字目かをの数値を並行して積むなどする。)

e. その他、括弧の対応プログラムに好きな改良を施してみよ。

中置記法の変換

先のアルゴリズムは括弧だけ扱って式のほうは無視していたので、今度は式を扱いましょう。私達が普段使う数式は、演算子を値 (部分式) の間に書きます。これを**中置記法** (infix notation) と呼びます。

```
3 * 2 + 4
3 + 2 * 4
```

中置記法は「演算子の結び付きが強いものを先に計算する」という慣習があったりして複雑です。たとえば上の例では「*」を先に計算しますね。ここで**後置記法** (postfix notation) を導入します。後置記法は、値の順番は中置記法と同じですが、演算子を「後に」書きます。その「後に」の規則は「その演算子に最も近い 2 つの値を計算して結果に置き換えることで、求める式が計算できる」ようにします。上の 2 つの例を後置記法にしてみましょう (後置記法は日本語に置き換えると読みやすいという説もあるのでそれもやっています)。

```
3 2 * 4 +  →  6 4 + → 10   「3 と 2 を掛けたものに、4 を足す」
3 2 4 * +  →  3 8 + → 11   「3 と、2 に 4 を掛けたものを、足す」
```

中置記法の変換

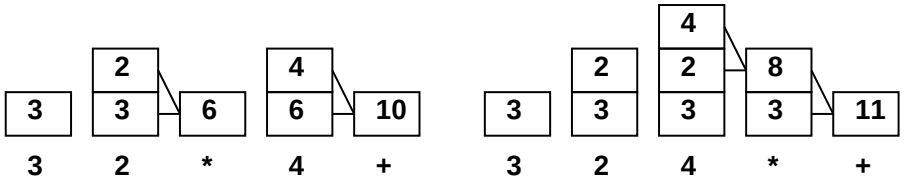

図 6: スタックを用いた後置記法の計算

そして、後置記法になった式はスタックを使って簡単に計算できます。計算のしかたは、(1)数値はそのままスタックに積み、(2)演算子は2つの値を降ろして来てその演算を行ない結果を積む、という動作を順にやるだけです(図6)。

中置記法の変換

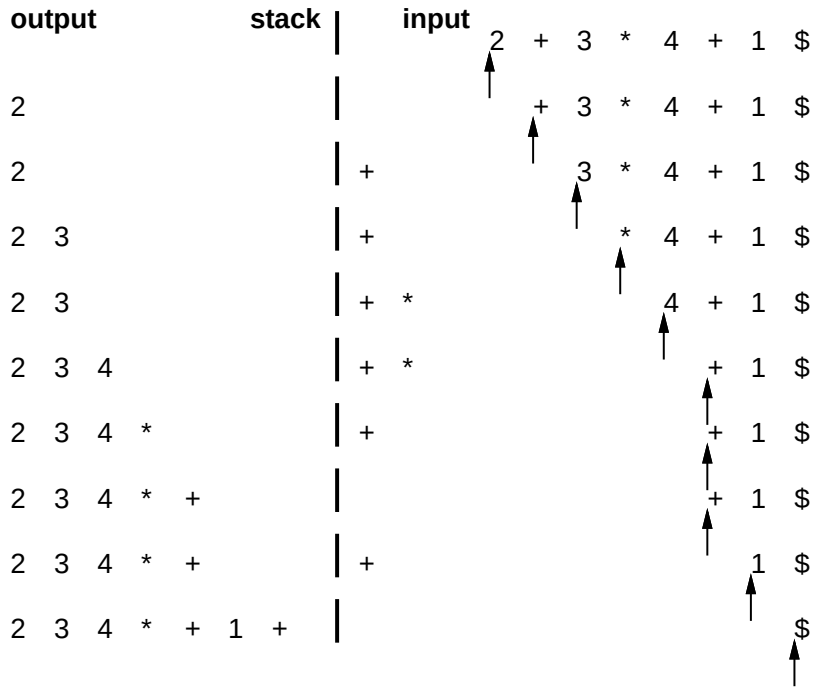

図 7: スタックを用いた中置後置変換

中置記法の変換

では、この変換を行なう方法を説明しましょう。まず、数値は順番が変わらないのでそのまま出力に送ります。演算子の場合の処理は次のようにします。

- スタックが空か、またはスタック先頭の演算子よりも入力先頭の演算子の方が結び付きが強いなら、入力先頭の演算子をスタックに積む。
- そうでない場合は、上記の条件が成り立たなくなるまで、スタックの演算子を降ろして出力に送る。

入力が終わりになったときも上記の2番目同様、残っている演算子は降ろして出力に送ります。これを行なうCのプログラムを示しましょう。演算子として「+」「-」を強さ1、「*」「/」「%」を強さ2、そしてべき乗のつもりの「^」を強さ3としました(実際のC言語にはべき乗演算子はなく、「^」はビット毎排他的論理和演算子なので注意)。

中置記法の変換

また、ここまでは「出力に送る」と書いて来ましたが、実際には変換をおこなう関数`postfix1`は入力の文字列と出力文字列を格納する文字配列(実際には文字へのポインタ)を受け取ります。「送る」というのはですから、出力文字ポインタの位置に文字を書き込み、文字ポインタは次の位置に進める、ということです。これはC言語では「`*u++ = 文字;`」という書き方で実現できます。文字列として扱えるようにするために、最後にナル文字を書き込む必要があります。

中置記法の変換

```
// postfix1.c -- convert infix to postfix (w/o parentheses)
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include "istack.h"
int operprec(int c) {
  switch(c) {
    case '+': case '-': return 1;
    case '*': case '/': case '%': return 2;
    case '^': return 3;
    default: return 0;
  }
}
void postfix1(char *t, char *u) {
  istackp s = istack_new(200);
  for(int i = 0; t[i] != '\0'; ++i) {
```

```
    char c = t[i];
    if(isdigit(c)) { *u++ = c; continue; }
    while(!istack_isempty(s) &&
          operprec(istack_top(s)) >= operprec(c)) {
      *u++ = istack_pop(s);
    }
    istack_push(s, c);
  }
  while(!istack_isempty(s)) { *u++ = istack_pop(s); }
  *u++ = '\0';
}
int main(int argc, char *argv[]) {
  char buf[200];
  for(int i = 1; i < argc; ++i) {
    postfix1(argv[i], buf);
    printf("%s => %s\n", argv[i], buf);
  }
```

```
  return 0;
}
```

中置記法の変換

なお、isdigit というのは ctype.h を include すると使えるようになる標準関数で、文字が数字 (0〜9) であるなら「はい」を返します。上のコードでは数字ならそれをコピーするだけでもう終わりなので、continue 文で次の周回に進んでいます。それ以外は演算子の場合で、そのロジックは上に説明した通りです。実行例を示します。

```
% ./a.out '1+2*3+4'
1+2*3+4 => 123*+4+
```

それでは単体テストを作ります。今回はそもそも、文字列が結果なので、expect_str を作るところからやります。文字列の比較にはライブラリの strcmp を使うので、string.h の include が必要です。strcmp は「等しいと 0 を返す」ので、文字列 OK と NG の位置がこれまでと反対です。

中置記法の変換

```
// test_postfix1.c --- unit test of postfix1
#include <ctype.h>
#include <stdio.h>
#include <string.h>
#include "istack.h"
(operprec, postfix1 here)
void expect_str(char *s1, char *s2, char *msg) {
  printf("%s '%s':'%s' %s\n", strcmp(s1, s2)?"NG":"OK", s1, s2, msg);
}
int main(void) {
  char buf[200];
  postfix1("1+2*3", buf); expect_str(buf, "123*+", "1+2*3 => 123*+");
  postfix1("2*3+1", buf); expect_str(buf, "23*1+", "2*3+1 => 23*1+");
  return 0;
}
```

中置記法の変換

演習 2 上の中置後置変換の例題をそのまま動かせ。さまざまな式を変換して動作を確認すること。終わったら次のことをやってみよ。単体テストすること。

a. 「2^3^4」のようにべき乗演算が連続している場合、現在は $(2^3)^4$ のように左から計算されるが、慣習としては $2^{(3^4)}$ のように右から計算されるべきである。そのように修正してみよ (その結果、他の演算も並んでいるときに右から計算されるようになったとしても、まあよいことにする)。

b. 上の中置後置変換プログラムは括弧に対応していない。括弧が使えるように修正してみよ。(ヒント: 括弧対応プログラムのように開き括弧を積み、閉じ括弧のところで対応する開き括弧が来るまで降ろして出力する。)

c. 後置記法に変換するだけでなく、その変換結果をもとにスタックで値を計算して式の値を求めるようにせよ。入力に現れる数は1桁だが、結果は2桁以上であってよい。

中置記法の変換

d. 計算機能に加えて変数への代入機能と変数参照機能を追加してみよ。この場合、入力は複数行にわたってよく、空行を入力すると終了するようにするのがよい (詳細は任せる)。

e. その他、中置後置変換プログラムに好きな改良を施してみよ。

文字列の扱い
ファイルの上下逆転

ここまでは整数を積むスタックを扱って来ましたが、たとえば実数値など、他の種類のデータを扱うこともももちろんできます。ここでは文字列を扱う例を見てみましょう。文字列は文字ポインタですから、まずポインタ値を扱える版のスタックを定義します。ヘッダファイルは次の通り。

ファイルの上下逆転

```
// pstack.h --- pointer type stack interface
#include <stdbool.h>
struct pstack;
typedef struct pstack *pstackp;
pstackp pstack_new(int size);
bool pstack_isempty(pstackp p);
bool pstack_isfull(pstackp p);
void pstack_push(pstackp p, void *v);
void *pstack_pop(pstackp p);
void *pstack_top(pstackp p);
```

扱う値がvoid*型になっただけですね。なお、実際には様々なポインタを扱うわけですが、それは後で見るように、void*との間でキャストして扱います。実装の方も示しましょう。

ファイルの上下逆転

```
// pstack.c --- pointer type stack impl. with array
#include <stdlib.h>
#include "pstack.h"
struct pstack { int ptr, lim; void **arr; };
pstackp pstack_new(int size) {
  pstackp p = (pstackp)malloc(sizeof(struct pstack));
  p->ptr = 0; p->lim = size; p->arr = (void**)malloc(size * sizeof(void*))
  return p;
}
bool pstack_isempty(pstackp p) { return p->ptr <= 0; }
bool pstack_isfull(pstackp p) { return p->ptr >= p->lim; }
void pstack_push(pstackp p, void *v) { p->arr[p->ptr++] = v; }
void *pstack_pop(pstackp p) { return p->arr[--(p->ptr)]; }
void *pstack_top(pstackp p) { return p->arr[p->ptr - 1]; }
```

ファイルの上下逆転

値がvoid*なので、それを並べて格納する配列は型としてはvoid**になることに注意してください。では、これを使ってファイルの上下逆転をやってみます。

```
// reversefile.c --- input a flie and output upside-down
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "pstack.h"
bool getl(char s[], int lim) {
  int c, i = 0;
  for(c = getchar(); c != EOF && c != '\n'; c = getchar()) {
    s[i++] = c; if(i+1 >= lim) { break; }
  }
  s[i] = '\0'; return c != EOF;
}
int main(void) {
  char buf[200];
```

```
  pstackp s = pstack_new(200);
  while(getl(buf, 200)) {
    char *t = (char*)malloc(strlen(buf)+1);
    strcpy(t, buf); pstack_push(s, t);
  }
  while(!pstack_isempty(s)) {
    char *t = (char*)pstack_pop(s); printf("%s\n", t); free(t);
  }
  return 0;
}
```

ファイルの上下逆転

getlは1行読み込み、文字列の最後にナル文字を入れます。そして最後にEOF(ファイルの終わり)でないかどうかを返します。[2]mainの方ではスタックを用意し、getlで読めたらそれを積みます…が、bufは次の行を読み込むのに使う領域ですから、文字列の長さ+1(ナル文字のぶん)の領域を割り当て、そこにコピーして、その領域のポインタを積みます。終わりまで積み終わったら、今度は順次降ろしながら出力しますが、出力したらその行はもう使わないのでfreeしています。動作例を示します。

```
% ./a.out
this is a pen.
that is a dog.
how are you?
^D   ← Ctrl-Dで終わりの印
how are you?
that is a dog.
this is a pen.
```

[2]なぜ for の先頭での変数宣言をしていないかというと、変数 c も i も for が終わった後でも参照するからです。

ファイルの上下逆転

演習 3 スタックに「現在格納している値の個数」を調べる機能と、「満杯であるかどうか調べる機能」を追加してみよ。単体テストを作成して確認すること。

行バッファの実装

エディタなどでの内部では、行の並びを扱い、任意位置での挿入や削除ができるようなデータ構造が必要とされます。1つの選択肢は連結リストを使うことですが、スタックを2つ「向かい合わせで」使うことで、そのようなデータ構造が作れます。その概要は次の通りです。

- 「現在位置」は2つのスタックの「間」にあるものと考え、2つのスタックはそれぞれ現在位置より上にあるものと、下にあるものが入っていると考える。
- 現在位置を1つ下や上に動かすことは、下のスタックから1つ降ろして上のスタックに積んだり、その逆をすることで行なえる。
- 現在位置の上に行を挿入するには、単にその行の内容を上のスタックに積めばよい。
- 現在位置の直後の行を削除するには、単に下のスタックから1行降ろせばよい。

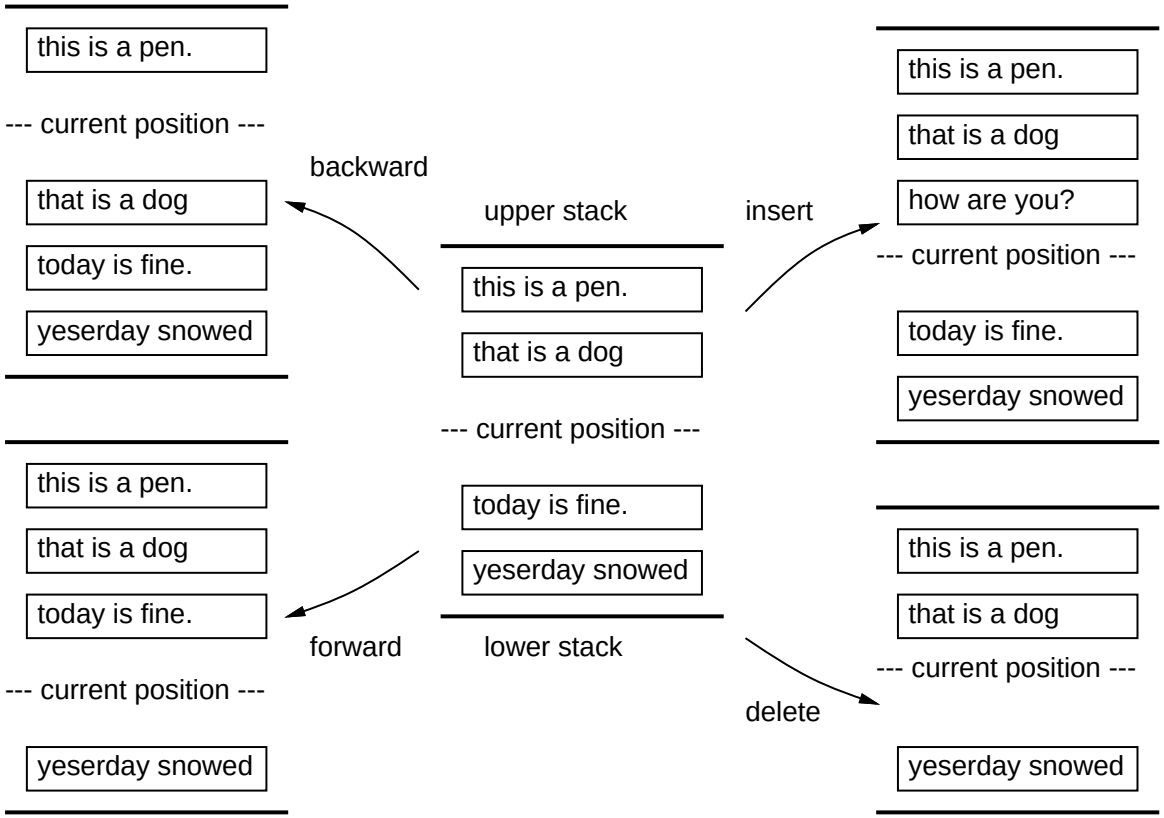

図 8: 2 つのスタックを用いた編集バッファ

行バッファの実装

演習4 スタックを2つ使って行バッファの機能を実現してみよ。行バッファもレコード型を使ってカプセル化できた方がよい。単体テストを作成して確認すること。

演習5 前問の行バッファを利用してテキストエディタを作成してみよ。機能は自由に設計してよい。

演習6 スタックを使って何か面白いプログラムを作れ。

本日の課題3A

「演習1」～「演習6」で動かしたプログラム1つを含むレポートを本日中(授業日の23:59まで)に提出してください。

1. 学籍番号、氏名、ペアの学籍番号(または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
2. プログラムどれか1つのソースと「簡単な」説明。
3. レビュー課題。提出プログラムに対する他人(ペア以外)からの簡単な(ただしプログラムの内容に関する)コメント。
4. 以下のアンケートの回答。

   Q1. スタックとその働きについて理解しましたか。

   Q2. 「かっこの対応検査」や「後置記法への変換」について納得しましたか。

   Q3. リフレクション(今回の課題で分かったこと)・感想・要望をどうぞ。

次回までの課題 3B

「演習 1」～「演習 6」(ただし 3A で提出したものは除外、以後も同様) の (小) 課題から選択して 2 つ以上プログラムを作り、レポートを提出しなさい。できるだけ複数の演習から選ぶこと。レポートは次回授業前日 23:59 を期限とします。

1. 学籍番号、氏名、ペアの学籍番号 (または「個人作業」)、提出日時。名前の行は先頭に「@@@」を付けることを勧める。
2. 1 つ目の課題の再掲 (どの課題をやったか分かればよい)、プログラムのソースと「丁寧な」説明、および考察 (課題をやってみて分かったこと、分析、疑問点など)。
3. 2 つ目の課題についても同様。
4. 以下のアンケートの回答。

   Q1. スタックがさまざまなことに役立つことを納得しましたか。

   Q2. 文字列の取り扱いはどれくらいできるようになりましたか。

   Q3. リフレクション (今回の課題で分かったこと) ・感想・要望をどうぞ。