プログラミング通論'19 ＃5 – 再帰呼び出しとその実装

久野 靖 (電気通信大学)

2019.4.20

今回は次のことが目標となります。

- 変数の可視範囲/存在期間と引数渡しについて理解する。
- 再帰呼び出しとその実装方法を理解する。

1 変数とその実現

1.1 変数の可視範囲

変数の**可視範囲** (**スコープ**、scope) とは、その変数の名前を指定して変数にアクセスすることができるコード上の範囲を言います。例として、図 1 を見てください。

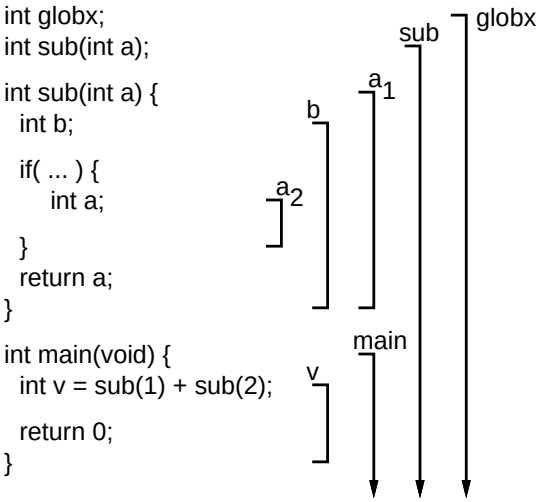


図 1: スコープの図解

C 言語ではこれまで見てきたように、変数や関数は宣言/定義した箇所から後ろで使えます。ですから、たとえばグローバル変数 `globx` がファイルの先頭で宣言されていれば、それはその先ファイルの終わりまでがずっと使える範囲つまりスコープです。関数 `sub` はプロトタイプ宣言があるので、それ以降がスコープです。`main` は定義しかないので、定義の先頭以降がスコープです。

では、ローカル変数はどうでしょうか。ローカル変数は関数内で使えるものですから、ローカル変数 `b` のスコープは関数 `sub` の中になります。また、パラメタ `a` も関数の先頭で宣言されているわけで、同様です。

この先がややこしいのですが、C 言語ではブロック (「{…}」で囲まれた範囲) の中で宣言された変数のスコープはそのブロックの終わりまでとなっています。ですから、内側の if のブロック中にある `a`(区別できるように添字をつけて a_2 としました) はそのブロックの末尾までがスコープです。ということは、パラメタ a_1 のスコープの中でこの範囲だけは「穴」があいているわけです。これを (内側の変数が外側の変数を「隠す」ことから) **シャドウ** (shadow) する、と言います。

関数末尾の return のところでは、内側の a_2 のスコープは終わっていますから、ここで返す値は a_1 の方ということになります。

main 内のローカル変数 i については、そこから関数の終わりまでがスコープになります。

1.2 変数の存在期間

変数の**存在期間** (**エクステント**、extent) とは、その変数が存在している時間的な範囲のことを指します。グローバル変数の存在期間はプログラムが始まった時点から終了する時点までずっとです。スコープから外れている間も (たとえばローカル変数で globx という名前のものを定義すれば、そのスコープでは外側のグローバル変数はシャドウされてスコープ外になります)、変数自体は存在し続けていることに注意。

そして、ローカル変数やパラメタは宣言された箇所 (パラメタでは関数の先頭) を実行する時点から、そのブロック (関数本体のブロックも含む) を出るところまでが、エクステントとなります。エクステントを出るところでは変数が無くなるのですから、そのあと再度エクステントに入った場合に、前の値が残っているというわけには行きません。

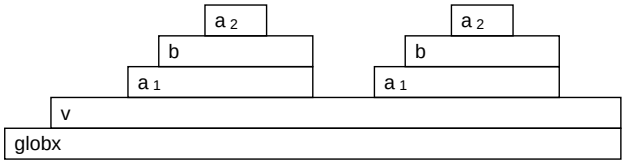


図 2: エクステントの図解

図 2 に先のコードのエクステントの推移を図示しました。仮に main から sub を 2 回呼び出したものとしています。そのため、sub のパラメタやローカル変数は 2 回に分けて現れます。

なお、a_2 については「if 文の条件が成り立って中を実行した時にはじめて」存在するようになることに注意。このように、エクステントは動的な (実行してみて始めて分かる) 性質を持ちます。一方、スコープは「どの箇所ではどの変数が見えているか」ということなので静的な (実行しなくても/コンパイル時に分かる) 性質を持ちます。

ところで図 2 を見ると、内側の変数ほど後で現れ、先に無くなるので、ちょうどスタックに変数を積んだり降ろしたりしているように見えます。これはブロックが入れ子構造になっていることから自然にそうなるのです。そして実際、変数の領域の管理は言語処理系の内部ではスタックを用いて行なうことが普通です。

1.3 実行時スタックと戻り番地

前節末で述べたように、言語処理系の内部では変数等をスタックで管理します。このスタックを**実行時スタック** (runtime stack) と呼びます。その様子をもう少し詳しく見て見ましょう。

図 3 のように、プログラムが実行を開始したところでは、main の変数 i だけが実行時スタックに割り当てられています (そのほかに空白や灰色の箇所もありますが、これは制御情報として使われています)。そして、そこから sub が呼ばれると、実行時スタック上に sub が使う 3 つの変数 (パラメタを含む) の領域が取られます。そして、sub から戻るとまた最初と同じになり、再度 sub が呼ばれるとまた 3 つの変数の領域が取られます。

なんだ、いつも同じ場所では、と思ったかも知れませんがそうではないのです。仮にそのあとで main が sub2 という関数を呼び、その関数が sub を呼んだとすると、sub2 の変数 (ここでは x、y としています) が取られ、その上に sub の 3 つの変数が取られます。スタックなので、常に最後に割り当てられた領域が最初に開放されるという形で進んで行きます。

この、1 つの関数実行に対応するスタック上の領域のことを**スタックフレーム** (stack frame) と呼びます。スタックフレームには、その関数のローカル変数と制御情報が含まれます。しかし制御情報っ

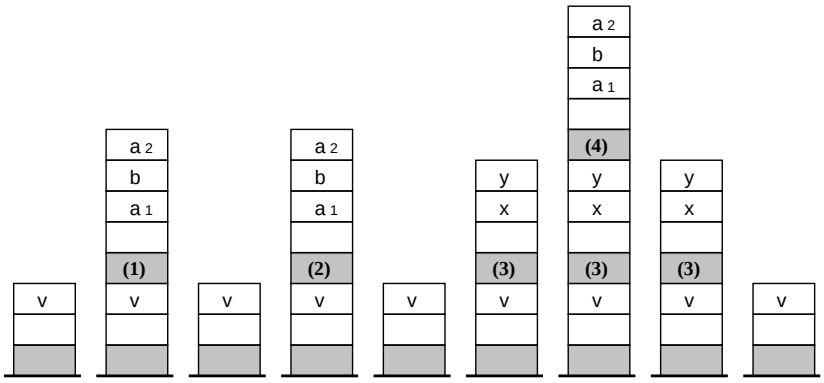


図 3: 実行時スタックの推移

て何でしょうか? いくつかありますが、ここでは最も重要な**戻り番地** (return address) だけ説明しておきます。

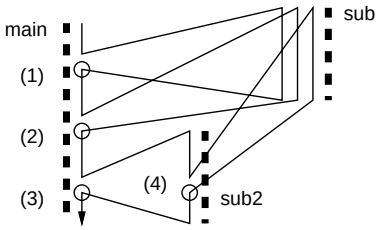


図 4: 実行の一筆描きと戻り番地

プログラムが CPU 命令の列に翻訳されて実行されるとき、命令はメモリ上に順番に並べられていて、それを CPU が上から順に実行していきます。分岐や反復のところはまた別ですが、ともかく 1 命令ずつ「一筆描き」のようにして実行が進みます。

図 4 では 3 つの関数の命令列を太い点線で示しています。ここで `main` から実行開始し、`sub` を呼ぶと実行は `sub` の先頭に移り、その先頭から順に実行が進みます。そして `sub` の最後まで来ると… 今度は、`main` 中のさっき `sub` を呼んだ命令の「次の」命令に戻り、そこから実行を続けます。もう 1 回 `sub` を呼んだ時も同様ですが、戻って来るのはその呼んだ命令の次ですから、さっき戻った位置とは違います。

そして次に `sub2` を呼び、`sub2` から `sub` を呼ぶと、今度は戻って来るのは `sub2` の中の「呼んだ次の」命令です。そして `sub2` から戻る位置は `main` 中の—tt sub2 を呼んだ次の命令です。

ということは、呼んだ時にこれらの「次の」命令の位置 (図では○で表しています)、というかメモリ番地を覚えておく必要があるわけです。これが戻り番地です。そしてそれをどこに覚えておくかというと、実行時スタックに覚えておくわけです。それが先の図では灰色で表された箇所になります (それぞれの位置に対応した番号が記入してあります)。[1]

1.4 引数と引数渡し

ここで**引数** (parameter ないし argument) についてもう少し見ておきます。関数定義の先頭には**仮引数** (formal parameter) のリストがあり、ここで引数の個数と型、およびそれぞれの引数の名前が定義されています。そして、実際に関数を呼び出すところでは、それぞれの仮引数に渡す式のリストを与えます (図 5 左)。これを**実引数** (actual parameter) と呼びます。

C 言語をはじめ多くの言語では、実引数のそれぞれの式の結果 (値) が、対応する仮引数の初期値として渡されます。関数の中では、仮引数は初期値の格納されたローカル変数としてふるまいます。

[1] `main` にも対応する戻り番地があることに気がついた方がいるかも知れません。これは、OS がプログラムを起動して `main` を呼び出すコード中の番地で、そこへ戻ると単にプログラムを終了させます。

これを**値渡し** (call by value) の**引数機構** (parameter mechanism) と呼びます。

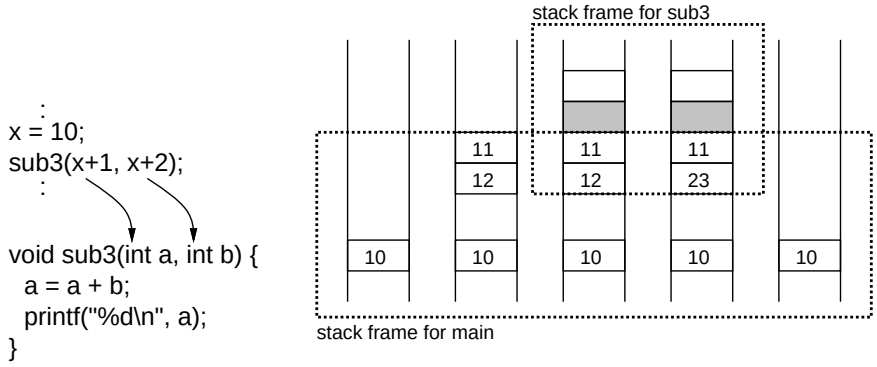


図 5: 引数と値渡しの実装

これがどのように実装されるかを見てみましょう (図 5 右)。`main` の中で実引数の値を計算し、スタックに置きます。続いて `sub3` を呼ぶと、`sub3` の中では戻り番地よりも下にある引数を他のローカル変数と同じように読み書きして計算を進めます。つまり引数は関数を呼ぶ側のスタックフレームと呼ばれた関数のスタックフレームが共有する部分になるわけです。このような、関数呼び出し時に何をどこに置いてどのように渡すかの取り決めを**呼び出し規約** (calling convention) と呼びます。呼び出し規約には**返値** (return value) の受け渡し方も含まれますが、返値は 1 つだけなので特定の CPU レジスタに入れて戻るやり方が普通です)。[2]

C++や Pascal など 一部の言語では引数機構として**参照渡し** (call by reference) を使うこともできます。参照渡しでは、実引数として変数を書いた場合、そのアドレスが渡され、関数側で仮引数に代入すると、対応する実引数の変数が変更されるようになります。関数から複数の値を返したい場合にはこのような機構が便利です。しかし C 言語には値渡ししかないので、ポインタの「値」を渡して間接参照で代入するわけです (それを自動的にやってくれるのが参照渡しだと癒えます)。

たとえば、2 つの実数変数の値を交換する関数 `dswap(double *x, double *y)` を書くことを考えます。変数の値を書き換えるには左辺値が必要ですから、ポインタを受け取りって間接参照で書き換えるわけです。

```
// dswaptest.c --- demonstration of dswap.
#include <stdio.h>
#include <stdlib.h>
void dswap(double *x, double *y) {
  double z = *x; *x = *y; *y = z;
}
int main(int argc, char *argv[]) {
  double a = atof(argv[1]), b = atof(argv[2]);
  printf("a = %g, b = %g\n", a, b);
  dswap(&a, &b);
  printf("a = %g, b = %g\n", a, b);
  return 0;
}
```

実行例は次の通り。

[2] スタックを介して渡とメモリの出し入れが必要となり速度が出ないので、最近のシステムでは引数も多数ある CPU レジスタに入れて渡す呼び出し規約が使われます。その場合、それをメモリに格納する位置 (つまり局所変数としての場所) は戻り番地より上に用意します。

```
% gcc8 dswaptest.c
% ./a.out 8 3
a = 8, b = 3
a = 3, b = 8
```

テストケースも示しましょう。実数は一般に計算誤差があるため、「等しい」で比べるのではなく「差 (の絶対値) がいくつ未満」でチェックするようにしています。ここでは許容誤差は「1.0×10^{-10}」としました。

```
// test_dswap.c --- unit test for dswap.
#include <math.h>
#include <stdio.h>
(dswap here)
void expect_double(double a, double b, double d, char *msg) {
  printf("%s %.10g:%.10g %s\n", (fabs(a-b)<d)?"OK":"NG" , a, b, msg);
}
int main(void) {
  double a = 3.14, b = 2.71; dswap(&a, &b);
  expect_double(a, 2.71, 1e-10, "a should be 2.71");
  expect_double(b, 3.14, 1e-10, "b should be 3.14");
}
```

実行例は次の通り。

```
% gcc8 test_dswap.c
% ./a.out
OK 2.71:2.71 a should be 2.71
OK 3.14:3.14 b should be 3.14
```

演習 1 ポインタと間接参照を使って、次のような関数を作れ。単体テストも作成すること。

a. `void rotate3(double *a, double *b, double *c)` — 3 つの実数変数のアドレスを受け取り、1 番目の値を 2 番目に、2 番目の値を 3 番目に、3 番目の値を 1 番目にと「順繰りに」転送する。

b. `void topolar(double x, double y, double *rad, double *theta)` — 2 次元直交座標の位置 (x, y) を受け取り、極座標形式に変換した結果をアドレスの渡された 2 つの実数変数に格納する。(ヒント: 「`man atan2`」はやってみた方がよい。)

c. `void normalize(double *x, double *y, double *norm)` — 2 次元ベクトル (x, y) を受け取り、正規化する (ノルム $x^2 + y^2$ を 1 にする)。元のベクトルのノルムを 3 番目のパラメタで渡された変数に返す。

d. `void divide(int a, int b, int *q, int *r)` — 整数 a を整数 b で割った時の商 q と余り r を返す。$a = b \times q + r$ であり、なおかつ a の符号 (正負) と q の符号 (正負) は一致すること (これは C 言語の除算、剰余とは違っている)。b が 0 のときは q も r も 0 とすること。

2 再帰呼び出しとその特性

2.1 再帰呼び出し

再帰呼び出し (recursive call) とは、ある関数が直接または間接に自分自身を呼び出すことをいいます。数学ではしばしば、再帰的な定義が使われますが、これをそのままコードに直すと再帰呼び出し

になります。以下は数学の定義に読めますね (ただし x、y は正の整数とします)。

$$gcd(x,y) = \begin{cases} x & (x = y) \\ gcd(x-y,\ y) & (x > y) \\ gcd(x,\ y-x) & (x < y) \end{cases}$$

それをそのまま C 言語のプログラムにするわけです。

```
int gcd(int x, int y) {
  if(x == y) {
    return x;
  } else if(x > y) {
    return gcd(x-y, y);
  } else {
    return gcd(x, y-x);
  }
}
```

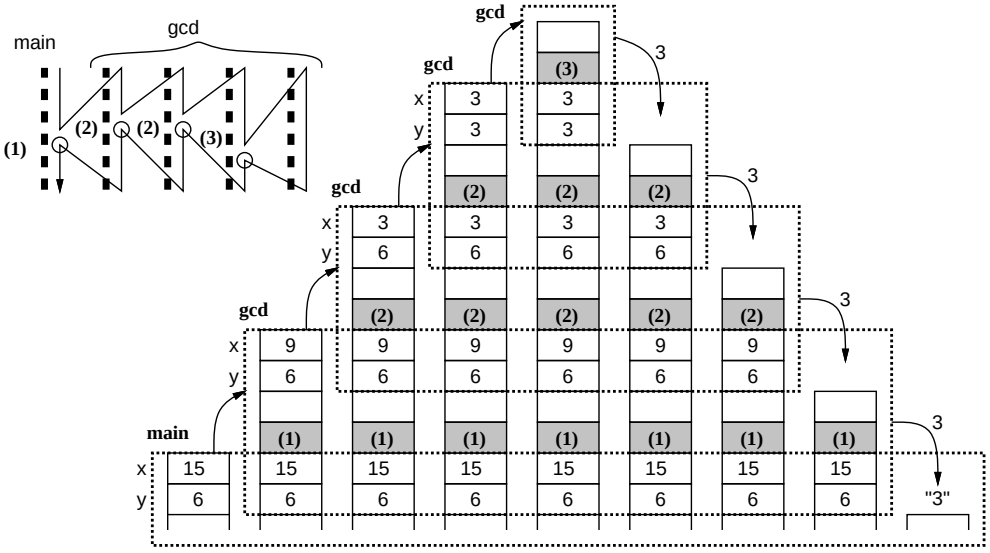


図 6: GCD の呼び出しと実行スタックのようす

それでは、再帰呼び出しはどのようにして実装されているのでしょう。実は、先に説明した実行時スタックを用いれば、そのままで再帰呼び出しは実装できます。例として、先の gcd を main から「gcd(15, 6)」のように呼び出した場合の実行時スタックの変化を図 6 に示します。

gcd のコードは実際には 1 つだけですが、ここでは分かりやすいように必要な個数コピーして次々に呼び出しているように描いてあります。gcd はローカル変数を持たず、パラメタ x と y は戻り番地より下に積まれて渡されてくることに注意。そして、戻り番地としては (1)main に戻るところ、(2)x が大きかった場合の再帰呼び出しから戻る、(3)y が大きかった場合の再帰呼び出しから戻る、の 3 通りがあることにも注意してください。

2.2 再帰の考え方

再帰呼び出しを数学の再帰的定義から考えることもできますが、もっと直接的な考え方を紹介しましょう。それは、「自分がやるべき仕事を少し簡単化して、自分の分身に頼む」というものです。そうやって簡単化していくと、最後は「頼まなくてもすぐできる」ようになるので、それは直接やります。

例題として**ハノイの塔** (tower of Hanoi) を取り上げましょう。これは図 7 のように棒の立った台座が 3 つ (A、B、C) と、何枚かの大きさの異なる円盤 (棒が入る穴つき) から成るパズルです (図では 3 枚)。円盤は小さいものから順に 1〜N の番号がついていて、最初は左上のように、A の台座に下から大きい順に積んであります。

それでパズルですが、円盤をいちどに 1 枚だけずつ他の台座に動かすことを繰り返して、最終的に右上のように B の台座にそっくり移してください。ただし、途中のどの段階でも「小さい円盤の上により大きい円盤を載せることはできません」。この 3 枚の場合であれば、実線の矢印のように順に動かして行けばできます。

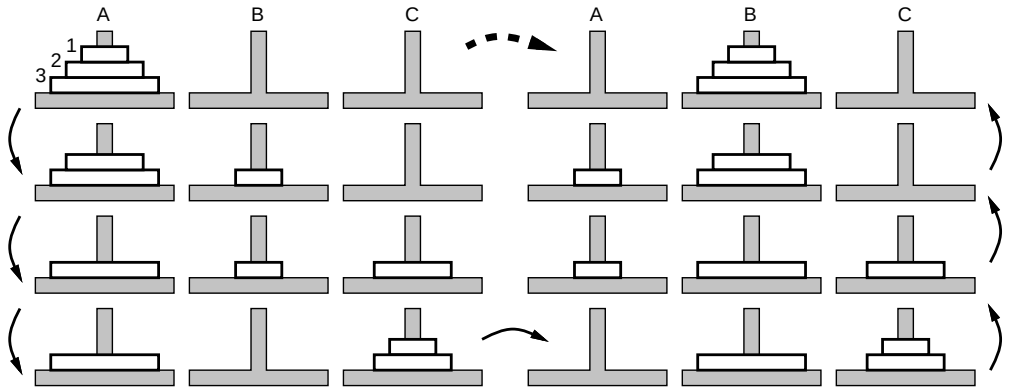


図 7: ハノイの塔

この動かし方を出力するプログラムを動かした様子を見ましょう。コマンド引数で円盤の枚数を与えます。

```
% ./a.out 3
move disc 1 from A to B.
move disc 2 from A to C.
move disc 1 from B to C.
move disc 3 from A to B.
move disc 1 from C to A.
move disc 2 from C to B.
move disc 1 from A to B.
```

これはどのように再帰を使って表せるでしょうか。次のように考えてください。

- C を作業場所として使いつつ A から B に K 枚の円盤を移すには、
- B を作業場所として使いつつまず A から C に K-1 枚の円盤を移し (※)、
- K の円盤を A から B に移し、
- B を作業場所として使いつつ C から A に K-1 枚の円盤を移せば (※) よい。

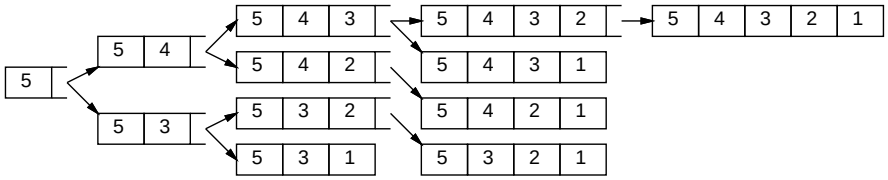
ここで (※) の 2 箇所は自分の分身に丸投げしていますが、問題が少し簡単になっているので (円盤の枚数が 1 枚だけ少ない)、これで大丈夫です。ポイントは、K の円盤が一番大きいのですから、それ以外の円盤はその上に載せて大丈夫なので、(※) の中で K の載っている A や B を作業場所に使ってよいというところです。あと、このままだと再帰が堂々めぐりですが、K が 1 になったらその上には円盤は載っていないので、直接行き先に移せます。

では、これをプログラムにしたものを見てみましょう。先の考え方をそのままコードにしていることが分かります (実行例は上に示しました)。

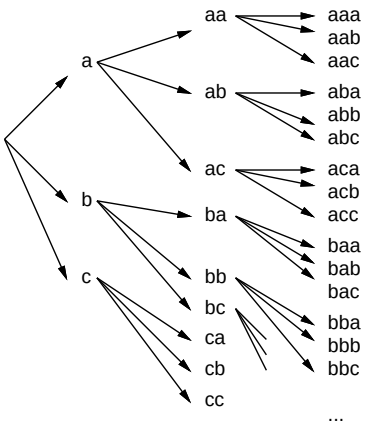
```
// hanoi.c --- tower of hanoi
#include <stdio.h>
#include <stdlib.h>

void hanoi(int k, int x, int y, int z) {
  if(k == 1) {
    printf("move disc %d from %c to %c.\n", k, x, y);
  } else {
    hanoi(k-1, x, z, y);
    printf("move disc %d from %c to %c.\n", k, x, y);
    hanoi(k-1, z, y, x);
  }
}
int main(int argc, char *argv[]) {
  hanoi(atoi(argv[1]), 'A', 'B', 'C'); return 0;
}
```

演習 2 「正の整数 n から始まる 1-2 減少列」とは、「最初が n、最後が 1 の減少数列で、隣り合う要素の差が 1 または 2 であるようなもの」である。たとえば 5 から始まる 1-2 減少列は「5, 4, 3, 2, 1」「5, 4, 3, 1」「5, 4, 2, 1」「5, 3, 2, 1」「5, 3, 1」の 5 つある。n を与えて n から始まる 1-2 減少列をすべて表示するプログラムを作れ。

ヒント: 配列 a に n から始まる 2-1 減少列を作るには、まず a の先頭に n を入れる。次に、a+1(a の次の要素から始まる配列) に n-1 から始まる 1-2 減少列と n-2 から始まる 1-2 減少列を作ってそれぞれ打ち出せばよい。再帰の終わりは n が 1 だったとき打ち出して戻ればよいが (0 なら行きすぎなので何もしない)、そのとき「全体を打ち出す」ためには、一番最初の配列の先頭が分かる必要がある (グローバル変数にするか、これもパラメタで受け渡す)。そして、先頭から現在の位置まで、ないし 1 を入れた位置まで打ち出す。

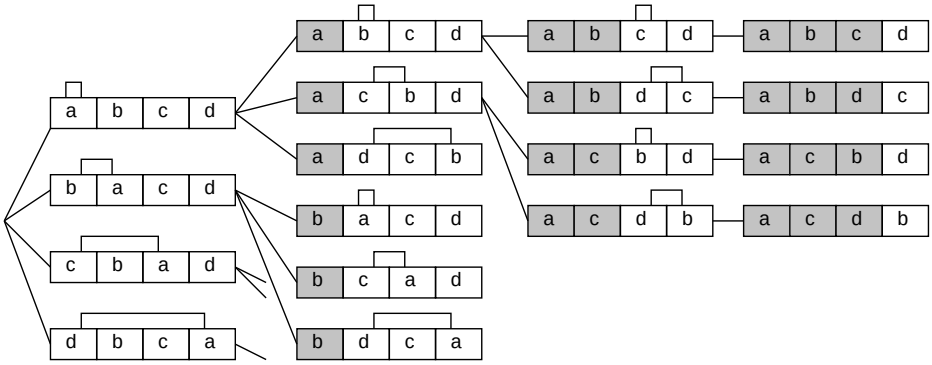


演習 3 文字の配列とその要素数、および生成する文字列の長さを与えて、指定された文字のあらゆる組み合わせで指定された長さの文字列を表示するプログラムを作れ。たとえば「abc」で長さ 3 なら、$3^3 = 27$ 通り、長さ 4 なら $3^4 = 81$ 通りの文字列を打ち出すことになる。



演習 4 文字の配列とその要素数を与えて、現れる文字の全ての順列を打ち出すプログラムを作れ。たとえば「abc」であれば「abc, acb, bac, bca, cab, cba」の 6 つが打ち出される。

ヒント: たとえば「abcd」であれば、まず最初が a の場合、b の場合、c の場合、d の場合を配列に用意する。それには、1 番目と 1、2、3、4 番目をそれぞれ交換すればできる。そのあと、配列の次の位置と、長さとして 1 減らした値を渡して自分自身を呼び出せば、それぞれの場合についてすべての順列を生成できる。この場合も最後に長さ 1 になったところで打ち出すためには、元の配列の先頭が必要となる。あと、再帰から戻ったところで交換した値を再度交換して元に戻さないとごちゃごちゃになる。必ず「元の状態に戻してから終わる」必要があるのに注意。



演習 5 順列生成プログラムを利用して自分の名前のアナグラムを生成せよ。ただし、ローマ字として読めるものだけを打ち出すこと。

ヒント: 順列プログラムは変更せず、最後に打ち出すところで「文字列がローマ字として正しいか」チェックするのがよい。[3]

演習 6 再帰を活用した自分の好きなプログラムを作れ。

2.3 再帰を使った深さ優先探索

再帰呼び出しはスタックによって実装されているので、スタックと同じ操作を行うことができます。具体的には「積む」ところで自分自身を呼び、「降ろして処理する」ところで普通に処理をして戻ればよいのです (処理した結果「積む」場合は再帰呼び出しします)。具体例として、前回やった図 8 の探索を再帰でやってみましょう。

`railmap.h` は今回は構造体のフィールドを変更しています。`prev` は、この節まで来たときの「1 つ前」の節番号を入れておき、あとで経路をたどれるようにします。`visit` は「この節を既に通ったか否か」を示すのに使います。[4]

```
// railmap2.h -- a railroad map (fields added)
#include <stdbool.h>
struct node { char *name; int num, edge[5], prev; bool visit; };
struct node map[] = {
  { "yokohama", 3, { 1, 3, 4 }, -1, false },    // 0
  { "kawasaki", 3, { 0, 2, 5 }, -1, false },    // 1
  { "shinagawa", 3, { 1, 3, 9 }, -1, false },   // 2
  { "osaki", 3, { 0, 2, 6 }, -1, false },       // 3
  { "hachiouji", 2, { 0, 5 }, -1, false },      // 4
```

[3] このように、候補を生成してみて OK かどうかチェックして使用するようなプログラムのことを generate-test 型のプログラムと呼びます。

[4] 前回はフィールド `dist` に距離と通ったか否かの役割を兼ねさせていましたが、今回は距離は節に格納しません。

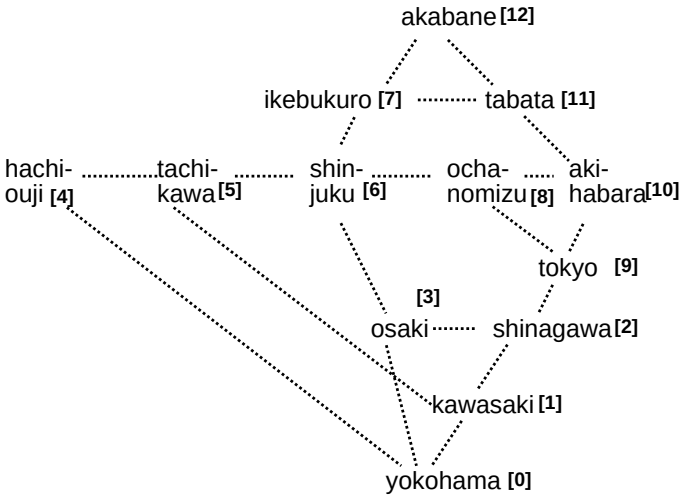


図 8: とある鉄道路線図 (再掲)

```
  { "tachikawa", 3, { 1, 4, 6 }, -1, false },   // 5
  { "shinjuku", 4, { 3, 5, 8, 7 }, -1, false }, // 6
  { "ikebukuro", 3, { 6, 11, 12 }, -1, false }, // 7
  { "ochanomizu", 3, { 6, 9, 10 }, -1, false }, // 8
  { "tokyo", 3, { 2, 8, 10 } , -1, false },     // 9
  { "akihabara", 3, { 8, 9, 11 }, -1, false },  // 10
  { "tabata", 3, { 7, 10, 12 }, -1, false },    // 11
  { "akabane", 2, { 7, 11 }, -1, false },       // 12
};
```

では、コードの方を示しましょう。重要なポイントですが、1 つ辺を進むごとに「どこから来たか」が分かるように prev に現在の節の番号を入れてから再帰呼び出しをしますが、戻って来たら prev の値を元に戻します。そのため、入れる前に変数 p に元の値を保管してあります。このように、再帰を使いながらデータ構造を書き換えるときには、「終わったら元に戻す」のが原則です。

```
// searchrec.c --- search path with recursion
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "railmap2.h"

void search(int cur, int goal, int len) {
  if(map[cur].visit) { return; }
  if(cur == goal) { printf("len = %d\n", len); return; }
  map[cur].visit = true;
  for(int i = 0; i < map[cur].num; ++i) {
    int k = map[cur].edge[i], p = map[k].prev;
    map[k].prev = cur; search(k, goal, len+1); map[k].prev = p;
  }
  map[cur].visit = false;
}
int main(int argc, char *argv[]) {
```

```
  int from = atoi(argv[1]), to = atoi(argv[2]);
  search(from, to, 0); return 0;
}
```

では、試しに品川から新宿まで何ステップで行けるかを調べます (あらゆる経路を調べることに注意)。

```
% ./a.out 2 6
len = 4
len = 5
len = 6
len = 3
len = 5
len = 5
len = 2
len = 3
len = 6
len = 7
len = 4
len = 5
len = 6
```

距離しか表示されないので分かりにくいですが、番号の若い節から順に試すので図と照合すればどこを通っているか分かります。最初の「4」は「品川→川崎→横浜→大崎→新宿」で、次の「5」は「品川→川崎→立川→横浜→八王子→立川→新宿」のようですね。

演習 7 上のプログラムをそのまま動かして動作を確認し、OK なら次のことをやってみなさい。

a. 距離 (通った辺の数) だけが表示されるのでは分かりにくいので、実際に通っている経路を表示するように改良する。(ヒント: `prev` にどこを通って現在の節まで来たかが入っているので、これを順に出発点までたどって行けばよい。)

b. 全部の経路を表示するのでなく、最短の (または最長の) 経路 1 つだけを表示するように変更する。(ヒント: 1 つ見つかるごとに打ち出す代わりに、経路を配列などに保管しておき、最短や最長が更新されたら新しいものに入れ換えるようにして、最後に打ち出す。)

c. 現在の「距離」は単に通った辺の数だが、その代わりに実際の距離 (または所用時間) を用いて最短、最長などを調べられるように変更する。(ヒント: 各辺ごとに距離や所用時間のデータを保持するようにデータ構造を変更する。)

d. その他、自分が面白いと思う改良をおこなう。

本日の課題 5A

「演習 1」〜「演習 4」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. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 5a ファイル名`」で以下の内容を提出。
2. 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。
3. プログラムどれか 1 つのソースと「簡単な」説明。

4. 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。
5. 以下のアンケートの回答。

   Q1. 引数の渡し方と手続き呼び出しの実現方法について理解しましたか。
   Q2. 再帰呼び出しのプログラムがどのように動いているか理解しましたか。
   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 5B

「演習 1」〜「演習 7」(ただし 5A 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. sol または CED 環境で「`/home3/staff/ka002689/prog19upload 5b` ファイル名」で以下の内容を提出。
2. 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。
3. 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)。
4. 2 つ目の課題についても同様。
5. 以下のアンケートの回答。

   Q1. 再帰呼び出しのプログラムが書けるようになりましたか。
   Q2. 再帰呼び出しでグラフの探索ができることを納得しましたか。
   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。