

私がプログラミング言語演習Ⅱ（前半）を担当していたときの資料です。再帰関数やスタックの解説が含まれています。2006～2009年の古い資料の寄せ集めのため、第6回以降、課題番号の整合性が取れていません。

第1回 型と制御構造の復習（課題1～9）	1
第2回 テストデータと再帰呼出し（課題10～18）	5
PAD入門	9
第3回 再帰関数, 迷路（課題19～23）	13
第4回 迷路をたどる（課題24～27）	17
揭示資料	19
第5回 スコープ, ポインタ, スタック（課題28～36）	20
第6回 中間試験（課題34～42）	24
第7回 迷路プログラムとデータ構造（課題39～40）	26
レポート表紙	30

---

## プログラミング言語演習Ⅱ 第1回

### 型と制御構造の復習

---

プログラム 1-1.c は階乗を求めるプログラムである。このプログラムでは、階乗を計算する関数 `fact()` の動作を調べるために、無限に `fact()` を呼び出すようになっている。プログラムを終了するには、`CTL-C` を入力する。このように関数を無限に呼び出すようなプログラムは、関数のデバッグや動作確認するには適している。

**課題1** 1-1.c をコンパイルして実行せよ。ただし、実行形式のファイル名は 1-1 とせよ。1-1.c に含まれる関数 `fact()` は、引数と戻り値は共に `int` 型である。引数 `n` をいろいろ変えて 1-1 を実行することにより、関数 `fact()` が正しく計算することができる `n` の最大値を求めよ。( `int` 型変数の最大値はいくつだろうか。) また、関数 `fact()` は、`n` の値が大きすぎると間違った計算結果を出力する。その理由を考えよ。また、`n` の値と間違った計算結果の関係を考えてみよう。

**課題2** 1-1.c の関数 `fact()` の引数と戻り値を共に `long` 型になるように修正せよ。修正したプログラム名を 1-1L.c とせよ。そして、課題1と同じように、関数 `fact()` が正しく計算することができる `n` の最大値を求めよ。( `long` 型変数の最大値はいくつだろうか。)

**課題3** 1-1.c の関数 `fact()` の引数と戻り値を共に `unsigned long` 型になるように修正せよ。修正したプログラム名を 1-1UL.c とせよ。そして、課題1と同じように、関数 `fact()` が計算することができる `n` の最大値を求めよ。

プログラム 1-1.c の制御構造は `for` ループである。このプログラムは、`fact(5)` を計算するのに、`5*4*3*2*1` として計算するので下方向計算と呼ぶことにする。一方、プログラム 1-2.c の制御構造も `for` ループである。このプログラムは、`fact(5)` を計算するのに、`1*2*3*4*5` として計算するので上方向計算と呼ぶことにする。`for` 文ではなく、`do-while` 文、`while` 文を使って 1-1.c 及び 1-2.c と同等の動作をするプログラムを作成することを考える。作成したプログラムが、正しく動作することを確認せよ。すなわち、引数として、0, 1, 課題1で求めた最大値を入力し、計算結果が正しいことを確認せよ。自分が作成したプログラムが正しく動作すること、客観的に示すように努力せよ。

**課題4** プログラム 1-1.c を修正して、制御構造は `do-while` で、下方向計算で `fact()` を求めるプログラムを作成せよ。ファイル名は、1-1dw.c とせよ。また、正しく動作することを示せ。

**課題5** プログラム 1-2.c を修正して、制御構造は `do-while` で、上方向計算で `fact()` を求めるプログラムを作成せよ。ファイル名は、1-2dw.c とせよ。また、正しく動作することを示せ。

**課題6** プログラム 1-1.c を修正して、制御構造は `while` で、下方向計算で `fact()` を求めるプログラムを作成せよ。ファイル名は、1-1w.c とせよ。また、正しく動作することを示せ。

課題7 プログラム 1-2.c を修正して、制御構造は while で、上方向計算で fact() を求めるプログラムを作成せよ。ファイル名は、1-2w.c とせよ。また、正しく動作することを示せ。

## プログラムの制御構造

教科書では、C 言語の文と、それに対応するフローチャートが形式として示されている。構文、形式、頁の対応表を右に示す。

「構造化プログラミング」についての記述が教科書の159頁にある。少し大きめの課題になると、いきなりプログラミングを書くことはまずしない。まず、フローチャート等を使って、アルゴリズムを記述する。そして、そのアルゴリズムが正しいことを確認して、プログラムを作成する。フローチャートなどの図を使った表現は、分かりやすく、ミスも見つけやすい。

構文	形式	頁
if 文	4.1	112
if-else 文	4.2	117
switch 文	4.3	118
switch 文	4.4	120
多岐条件文	4.5	123
while 文	4.6	132
for 文	4.7	145
do-while 文	4.8	150

課題8 次のプログラムのフローチャートを示せ。課題1の 1-1.c、課題4の 1-1dw.c、課題5の 1-2dw.c、課題6の 1-1w.c、課題7の 1-2w.d、フローチャートから C 言語のプログラムを作成することは簡単だろうか？ また、デバッグするときにフローチャートは、どのように役に立つだろうか？

## 再帰関数

階乗  $n!$  の定義は、負でない整数  $n$  に対して、 $0!=1$ 、 $n!=n*(n-1)!$ 、と定義できる。関数としては、 $\text{fact}(0)=1$ 、 $\text{fact}(n) = n * \text{fact}(n-1)$ 、と定義できる。この関数をプログラムしたものが 1-3.c である。関数  $\text{fact}()$  のように、自分自身を呼出している関数を再帰関数と呼ぶ。問題が再帰的に定義されているならば、それを解く関数も再帰的な構造にした方が良い場合がある。1-3.c と 1-1.c を比較してみれば良く分かるだろう。fact(4) は次のように動作する。

```
fact(4)  → 4 * fact(3)
          → 4 * 3 * fact(2)
          → 4 * 3 * 2 * fact(1)
          → 4 * 3 * 2 * 1 * fact(0)
          → 4 * 3 * 2 * 1 * 1
          → 4 * 3 * 2 * 1
          → 4 * 3 * 2
          → 4 * 6
          → 24
```

課題9 プログラム 1-3.c の関数  $\text{fact}()$  を修正することによって、右のような出力を得られるようにせよ。ファイル名は、1-3p.c とせよ。

fact(4) -> 4 * fact( 3)
fact(3) -> 3 * fact( 2)
fact(2) -> 2 * fact( 1)
fact(1) -> 1 * fact( 0)
fact(0) -> 1
fact(4) = 24

```

/*
    学籍番号      氏名      ファイル名 1-1.c
    n の最大値
*/
#include <stdio.h>
void main(void)
{
    int n, fact( int );
    while(1)
    {
        printf("¥nN = ");
        scanf("%d",&n);
        printf("¥tfact(%d) = %d¥n", n, fact(n));
    }
}

int fact( int n )
{
    int v, i;
    v = 1;
    for( i = n ; i != 0 ; i--)
        v = v * i;
    return v;
}

```

```

/*
    学籍番号      氏名      ファイル名 1-2.c
    n の最大値
*/
#include <stdio.h>
void main(void)
{
    int n, fact( int );
    while(1)
    {
        printf("¥nN = ");
        scanf("%d",&n);
        printf("¥tfact(%d) = %d¥n", n, fact(n));
    }
}

int fact( int n )
{
    int v, i;
    v = 1;
    for( i = 1 ; i <= n ; i++)
        v = v * i;
    return v;
}

```

```
/*
    学籍番号      氏名      ファイル名 1-3.c
    n の最大値
*/
#include <stdio.h>
main()
{
    int n, fact( int );
    while(1)
    {
        printf("N = ");
        scanf("%d",&n);
        printf("%t\tfact(%d) = %d¥n",n,fact(n));
    }
}

int fact( int n )
{
    if( n == 0 ) return 1;
    else return n * fact( n - 1 );
}
```

---

## 第2回 プログラミング言語演習Ⅱ テストデータと再帰呼出し

---

### ■準備（具体的な操作は第1回を参照のこと。）

/home/EX/PRO2/2/を自分のホームディレクトリにコピーせよ。2-6.c だけである。

### ■提出物

プログラムリストは、提出するリストをなるべく1つのファイルにまとめて印刷して、提出すること。ファイルをまとめるには、cat コマンドを使っても良いし、gedit を使っても良い。テストデータは、右余白に記入すること。課題10～16は必須。余力があれば、課題17と18を提出せよ。ただし、課題10と11は各自で確認しておけば良い。

### ■前回の解説

プログラムで数値計算などをするとき、忘れてはならないことは、変数が扱える数値には範囲がある、ということである。扱える数値範囲を超えると、課題1の関数 fact() のように間違った計算をする。簡単なプログラムでも、数値計算をするときは、その計算結果が扱える数値範囲に収まっているかどうかを気を配る必要がある。数値範囲を超えてしまう場合は、アルゴリズムを再検討するか、正しい結果が得られる引数の範囲を明示する必要がある。型と扱える数値範囲については、教科書65頁「表3.4」、84頁「表3.11」を参照のこと。

繰り返し処理のための制御構造には、for 文、while 文、do-while 文がある。アルゴリズムに応じて適切な制御構造を使うとプログラムが作りやすくなる。課題4と5の do-while 文は適切かどうかを考えよ。さらに、課題9の階乗のプログラムのように、与えられた問題が再帰的ならばプログラムも再帰的に作成することができる。

---

## テストデータ

---

プログラム（関数）を作成したら、自分が作成した関数が正しく動作することを客観的に示すことが必要である。前回の課題では、引数として、0、1、課題1で求めた最大値を入力し、出力が正しいことを確認した。自分が作成した関数が正しく動作することを示すためには、いろんな引数で正しい関数値が求まることを示せば良い。例えば、次のような表を用意すれば、ある程度は、正しく動作することを客観的に示すことができる。この表で、“正しい関数値”とはプログラムを作成するときに自分が想定した値であり、“実際の関数値”とは作成したプログラムで求めた値である。

関数名	引数	正しい関数値	実際の関数値	備考
fact()	0	1	1	定義 0!=1
//	1	1	1	1!=1
//				引数の最大値
//	-1	?	?	負数
...	...	...	...	...

このような表が示すデータを、一般的にテストデータと呼ぶ。プログラムを作成したら、同時にテストデータを作成するように心掛けよ。プログラム（関数）を改良したとき、テストデータがあれば簡単にその改良が正しいかどうかの判断が容易になる。また、テストデータには、関数 fact() に対する負数(-1)のように不適切な場合も含めると良い。どうしてだろうか。理由を考えてみよ。

**課題 10** 課題 1 に対する次のテストデータの表を完成させよ。正しい計算結果が得られないような引数も、入力される可能性があるならばテストデータに含めておくといい。他に考えられるテストデータがあれば、表の空欄に記入せよ。関数 `fact()` を使う上での注意事項を考えよ。

関数名・課題	引数	正しい関数値	実際の関数値	備考
課題 1	0	1		定義 $0!=1$
//	1	1		$1!=1$
//				引数の最大値
//				負の値
//				
//				
//				

**課題 11** 課題 10 で作成したテストデータを使って、課題 6～9 で作成した関数 `fact()` をテストせよ。実際の関数では、引数の値を調べて、不適切な引数ならばエラー表示などをする。関数が複雑になれば、このようなテストデータを作成することも大変になる。不適切な関数値を返す関数が含まれていると、最終的な計算結果が思わぬ値になることがある。

## 再帰呼び出し

提出時には、プログラムリストの右余白にテストデータを転記しておくこと。

**課題 12** フィボナッチ数列とは、次のように再帰的に定義することができる数列である。

$$\text{fib}(0)=1, \text{fib}(1)=1, n \geq 2 \text{ のとき } \text{fib}(n)=\text{fib}(n-1)+\text{fib}(n-2)$$

このフィボナッチ数列を求める再帰的プログラム 2-1.c を作成せよ。また、テストデータを次に示せ。関数名は `fib(n)` とし、プログラムは、無限に `fib()` を呼出すようにすること。関数 `fib()` の関数値と引数の型は、long 型とせよ。関数 `fib()` が正しく計算することができる  $n$  の最大値を求めてみよ。ただし、求めているとき、実行時間が長くなるようであれば適当に打ち切ること。

関数名	引数 $n$	正しい関数値	実際の関数値	備考
<code>fib()</code>				
//				
//				
//				
//				
//				

**課題 13**  $n$  個から  $k$  個を取り出す組合せの数は次のように再帰的に定義することができる。

$${}_nC_0 = {}nC_n = 1, \quad {}nC_k = {}_{n-1}C_k + {}_{n-1}C_{k-1}$$

この定義に従って組合せを計算する再帰的プログラム 2-2.c を作成せよ。関数名は `comb(n, k)`

とし、プログラムは無限に `comb()` を呼出すようにすること。関数 `comb()` の関数値と引数の型は、`long` 型とせよ。関数 `comb()` が正しく計算することができる `n` の最大値を求めよ。また、使用した関数 `comb()` のためのテストデータを次に示せ。欄が不足したら、別紙とせよ。

関数名	引数 <code>n</code>	引数 <code>k</code>	正しい関数値	実際の関数値	備考
<code>comb()</code>					
//					
//					
//					
//					

**課題 1 4** 階乗 `fact()` や組合せ `comb()` のような関数は、自分自身を呼出す構造を持っている。このような関数を自己再帰と呼んでいる。次の2つの関数は、相互再帰と呼ぶ構造をもっている。これら2つの関数の動作を調べるためのプログラム `2-3.c` を作成せよ。

```
int even( int n )
{
    if( n == 0 ) return 1;
        else return odd( n - 1 );
}
int odd( int n )
{
    if( n == 0 ) return 0;
        else return even( n - 1 );
}
```

**課題 1 5** 課題 1 4 のプログラムで、`even(5)` としたとき、どのように関数が実行されるかを解説せよ。図示しても構わない。リスト `2-3.c` の余白に記入せよ。

**課題 1 6** 乗算 `a*b` は、次のように定義することができる。

$$a * b = a + a * ( b - 1 ) \quad (b > 0 \text{ のとき})$$

この定義に従って乗算を計算するプログラム `2-4.c` を作成せよ。関数名は `mult(a,b)` とせよ。関数の関数値と引数の型は `long` とする。関数 `mult()` のためのテストデータを次に示せ。当然、関数 `mult(a,b)` は再帰関数とすること。

関数名	引数 <code>a</code>	引数 <code>b</code>	正しい関数値	実際の関数値	備考
<code>mult()</code>					
//					
//					
//					
//					
//					



**課題17** フィボナッチ数例は、次に示す関数 `fib2()` でも計算することができる。この `fib2()` の動作を確認するためのプログラム `2-5.c` を作成せよ。課題12のテストデータを使って、`fib()` と同じ計算をしていることを確認せよ。実行効率はどうだろうか？

```
int fib2( int n ) {
    return fib2_sub(1, 1, n);
}
int fib2_sub(int a, int b, int n) {
    if( n <= 0 ) return a;
    else if( n == 1 ) return b;
    else return fib2_sub(b, a + b, n - 1 );
}
```

**課題18** 配布したプログラム `2-6.c` を参考にして、課題12の `fib()` と課題17の `fib2()` の実行時間を表示するようなプログラム `2-7.c` を作成せよ。`n` の値に応じて、どのように実行時間が変化するのかを調べよ。

# PAD 入門

## 1 はじめに

PAD (Problem Analysis Diagram, 問題分析図) は、日立製作所中央研究所の二村良彦氏によって考案されたプログラミングのための図式である。日本人が考案した図式であり、国際標準にもなっている。PAD はフローチャート (流れ図) と似ているが、より見やすく、分かりやすい。

### 1.1 3つの基本形

すべてのプログラムは、次の3つの基本形だけの組み合わせで表現することができる。

- (1) 2つ以上の処理を時系列順に処理する「**接続**」
- (2) ある条件が成立する間は処理を繰り返す「**反復**」
- (3) ある条件に従って2つの処理の一方を選ぶ「**選択**」

PAD とフローチャートの対応を表1に示す。PAD では、縦方向が時間、横方向が処理の深さを表す。PAD は、フローチャートのようにフローを表す線が重なることがないため、全体の流れが把握し易いといえる。

PAD は、C言語が普及していない時代に考案された。このため、PAD に関する多くの書籍には、BASIC や Fortran といったプログラミング言語と PAD の対応は見つかるが、PAD と C 言語の対応を取り上げた書籍は少ない。次の節では、C 言語と PAD との対応について解説する。

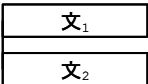

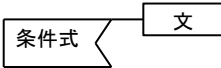
表1 3つの基本形と PAD<sup>1)</sup>

	適用例	フローチャート	PAD
接続	朝起きて、顔を洗って、朝食を採る。【順番に処理する。】		
反復	空腹の間中、食べる。【条件が成立している間、処理を繰り返す。】		
選択	食費が十分ならば、ステーキを食べ、そうでなければ、うどんを食べる。		

## 1.2 3つの基本形とC言語

3つの基本形について、PADとC言語の対応を表2に示す。反復は while 文に対応し、選択は if 文に対応する。

表2 3つの基本形とPAD

基本形	PADでの表記	C言語での制御構造	備考
接続		文 <sub>1</sub> ; 文 <sub>2</sub> ;	箱を縦方向に並べて、左端を接続すればよい。
反復		while( 条件式 ) 文 ;	条件式の判定後、条件が成立しているならば文を実行する。前判定反復と呼ぶことがある。
選択		if( 条件式 ) 文 ;	箱の中の「条件式」は、気持ち上に寄せて書く。

## 1.3 PADとC言語

3つの基本形以外のPADを表3と表4に示す。後判定の文である do-while 文は、反復に対応する while 文と似ているので、使用するときには注意すること。プログラムとしては、do-while 文はあまり使用しない方がよい。基本は、前判定である while 文である。

C言語特有の制御構造である for 文と switch 文に対するPADは、私が適当に考えた。ただし、switch 文については、完全には対応できない。

表3 PADとC言語の対応（その1）

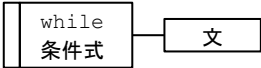
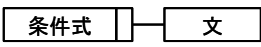

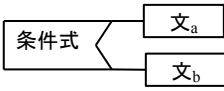
C言語での制御構造	PADでの表記	備考
while( 条件式 ) 文 ;		基本構造の反復と同じである。C言語の while 文との対応を明確に示したいとき使う。
do { 文 ; } while(条件式)		3つの基本構造の“反復”と似ているが、条件式が文の実行後に判定される。このため後判定反復と呼ぶことがある。
		C言語の do-while 文との対応を明確に示したいとき使う。
if( 条件式 ) 文 <sub>a</sub> ; else 文 <sub>b</sub> ;		if-else 文に相当する。

表4 PADとC言語の対応（その2）

C 言語での制御構造	PAD での表記	備考
<pre>if( 条件式<sub>1</sub> ) 文<sub>1</sub> ; else if( 条件式<sub>2</sub> ) 文<sub>2</sub> ;  else if( 条件式<sub>n</sub> ) 文<sub>n</sub> ; else 文<sub>n+1</sub> ;</pre>		多岐条件文を PAD で表記するには、基本構造の選択の考え方を発展させればよい。
<pre>for( 式<sub>1</sub> ; 式<sub>2</sub> ; 式<sub>3</sub> ; ) 文 ;</pre>		C 言語の for 文は複雑である。3つの基本構造だけで表記すると、このようになる。
		for 文用 PAD. for 文の式 <sub>2</sub> は、文を実行する前に判定されるので前判定反復の記号を使った。
<pre>switch( 式 ) { case 定数式<sub>1</sub> : 文<sub>1</sub> ; break; case 定数式<sub>2</sub> : 文<sub>2</sub> ; break;  default: 文<sub>n</sub> ; break; }</pre>		switch 文は定数式を条件式に置き換えれば if-else 文と同じである。しかし、各 case に break 文がない場合は、簡易的な表記方法はなく、3つの基本構造を使って表記することになる。

#### 1.4 PAD での関数定義

PAD には、C 言語での関数定義のための表記はない。しかし、関数定義も PAD で表現できた方が便利なので、図1のように関数定義を表すことにする。この表記は、私が適当に考えた。

C プログラム	PAD
<pre>int main( void ) {     int i ;     while( 1 )     {         printf("%ni = ");         scanf("%d", &amp;i ) ;         printf("%d", fact( i ) );     } } int fact( int n ) {     int i, fact;     fact = 1 ;     for( i = 1; i &lt;= n ; i++ )         fact *= i ;     return fact ; }</pre>	

図1 PAD による関数定義

## 1.5 PAD からプログラムへの変換

2次方程式を解くためのアルゴリズムを図2に示す。このPADには、変数 a, b, c の宣言、printf 文、scanf 文、文末に必要な “;” は含まれていない。また、変数の型や printf 文が重要ならば、この図に、`double a,b,c`, `printf("¥na=")`, `scanf("%lf",&a)`などを追加すればよい。特定のプログラミング言語に依存しなければ、それだけPADで表記したアルゴリズムは読みやすく、理解しやすくなる。逆に、C言語に依存すれば、それだけPADからC言語に変換が簡単になる。

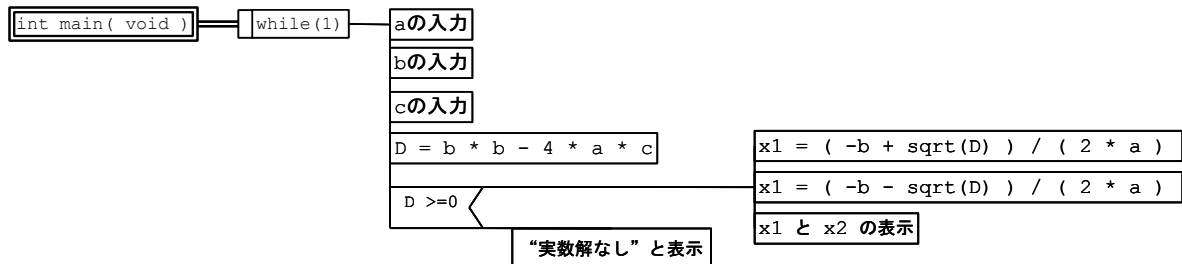


図2 2次方程式の解を表示するPADの例

PAD をプログラムに変換するには、ツリーウォーク（tree walk）と呼ぶ方法を使う。図3のように、PAD の先頭位置から右手で壁を触りながら PAD の木（tree）をたどる。そして、表2、表3にしたがって、プログラムに変換してゆけばよい。

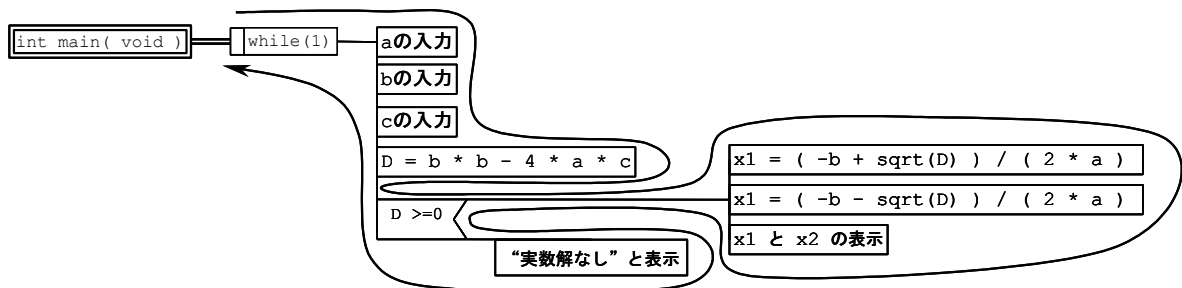


図3 ツリーウォーク

C言語に変換したプログラムを次に示す。

```
int main( void )
{
    double a,b,x1,x2,D;
    while( 1 )
    {
        printf("a=");
        scanf("%lf",&a);
        printf("b=");
        scanf("%lf",&b);
        printf("c=");
        scanf("%lf",&c);
        D = b * b - 4 * a * c ;

        if( D >= 0)
        {
            x1 = ( -b + sqrt( D ) ) / ( 2 * a ) ;
            x2 = ( -b - sqrt( D ) ) / ( 2 * a ) ;
            printf("x1=%lf, x2=%lf", x1 , x2 );
        }
        else {
            printf("実数解なし");
        }
    }
}
```

---

## 第3回 プログラミング言語演習Ⅱ 再帰関数，迷路

---

### ■準備

/home/info/EX/PRO2/3/を自分のホームディレクトリにコピーせよ。

### ■今回の内容

第2回の課題を引き続き行う。第2回の課題が終わったら、第3回の課題を行う。今回（第3回）の内容は、第2回で扱った再帰関数を引き続き扱う。

### ■提出物

第2回：課題10～18。課題17と18も提出すること。課題18については、調べたことは余白に書けばよい。

第3回：課題19～20を提出せよ。課題20はグラフを見せればよい。課題21～23は次回に持ち越しても良い。

### ■第2回の要点

我々が使用しているCコンパイラ（gcc）では、long と int は同じ大きさで、4バイト（32ビット）の大きさである。int 及び long で表せる数値は、 $-2^{31} \sim 2^{31}-1$ となる。このため関数 fact(n)のように、引数 n がある値を超えると正しい階上の値を計算できなくなる。

課題13で扱った関数 comb(n, k) では、引数が2つあるため、引数 n あるいは k の最大値を求めることが簡単ではない。引数 n と k の値によっては、“セグメンテーション違反です”と出力され、プログラムは終了してしまう。この原因については、課題19で扱う。

関数 fact(n) や関数 comb(n, k) は、プログラムは短いが、実行時間が長い。プログラム 1-1.c と 1-3.c の実行時間を比較してみれば分かる。同じ階上の計算をするのに、どうしてだろうか？

**課題19** 課題13で作成した関数 comb(n, k) が、どのように実行されるかを調べるために、long 型のグローバル変数 c を使って、次のように、関数 comb(n, k) の呼出し回数を表示するプログラム 3-1.c を作成せよ。配布したファイル 3-1.c を修正すればよい。

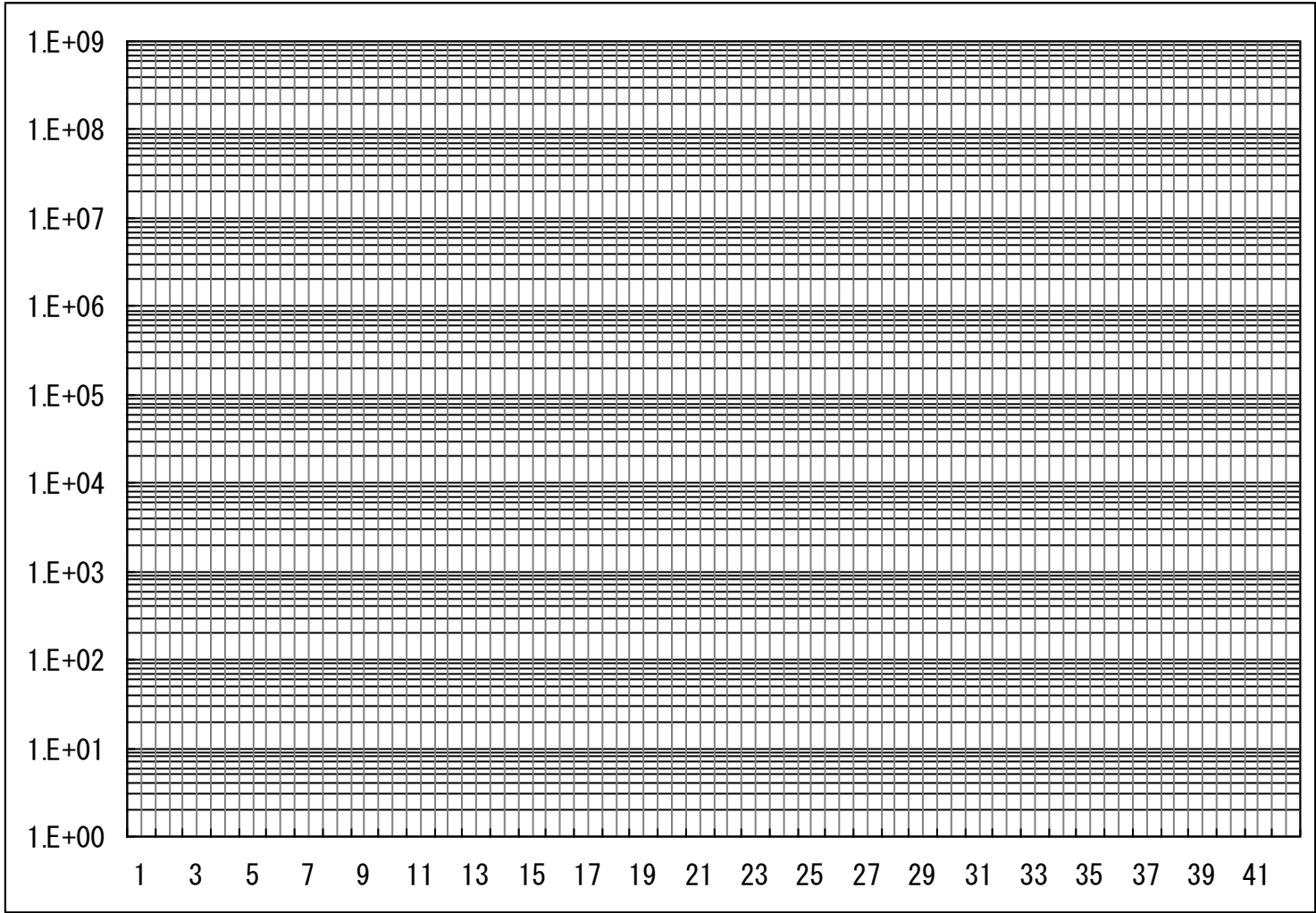
```
LONG_MAX = 2147483647
n, k = 10, 2
comb(10, 2)=45    times = 89
n, k = 10, 3
comb(10, 3)=120   times = 239
n, k =
```

引数 n を固定値とした場合と、引数 k を固定値とした場合とで、引数の値と実行回数の関係について調べてみよ。引数の値によって、“セグメンテーション違反です”と表示されて、プログラムが終了してしまうことがある。このとき、呼出し回数は、どのような値になってるだろうか？

**課題20** 課題12(2-1.c)で作成した関数 fib() と課題17(2-5.c)で作成した関数 fib2() は、同じフィボナッチ数列を求める関数であるが実行時間は大きく異なる。実行時間がどれだけ異なるかは、関数の呼出し回数を調べることによって知ることができる。関数 fib() と fib2() の呼出し回数を並べて表示するプログラム 3-2.c を作成し、次の表を埋めよ。また、片対数グラフに示せ。

n						
fib()の 呼出し回数						
fib2()の 呼出し回数						

n						
fib()の 呼出し回数						
fib2()の 呼出し回数						



## 迷路をたどる

迷路をたどるプログラムは、試行錯誤の繰返しである。すなわち、行けるところまで行き、行き止まりになったら、引き返し、新しい道を試して行く。配布したプログラムでは、その試行錯誤を一定の順序で進めることにより、出口を見つけている。

ここで、図3-1のような5×5の迷路を解くことを考える。

“■”は壁であり、“□”は道を示している。また、sは入口、Eは出口を表すことにする。図3-1で横方向をx、縦方向をyとし、迷路上での位置を(x, y)と表記する。すなわち、入口は(1, 1)、出口は(5, 5)となる。

入口から出口までの1つの経路は、(1, 1) → (2, 1) → (3, 1) → (3, 2) → (4, 2) → (5, 2) → (5, 3) → (5, 4) → (5, 5)である。プログラムでこのような経路を見つけることを考える。

このような迷路をプログラムで扱うには、迷路の周囲は壁“■”で囲まれていると考え、図3-2のように迷路を2次元の配列で表すことにする。このようにすると、入口(1, 1)は、右と下には移動できるが、上と左には移動できないと判断することが容易になる。データ構造を工夫することにより、アルゴリズムが簡潔になる例である。

図3-2の迷路をたどるには、現在の場所から、4方向を一定順で探って行き、移動できれば移動し、出口に達するまでこれを繰り返せばよい。探る順は、図3-3に示すように、右、下、左、上とする。もちろん、4方向すべてをたどれればよいので、上下左右でも構わない。

図3-2を2次元のint型配列maze[][]で表すと次のようになる。ここで、“■”を2、“□”を0とした。

```
int maze[7][7]={ 2,2,2,2,2,2,2 ,
                  2,0,0,0,2,0,2 ,
                  2,0,2,0,0,0,2 ,
                  2,0,0,0,2,0,2 ,
                  2,0,2,2,2,0,2 ,
                  2,0,0,0,0,0,2 ,
                  2,2,2,2,2,2,2 };
```

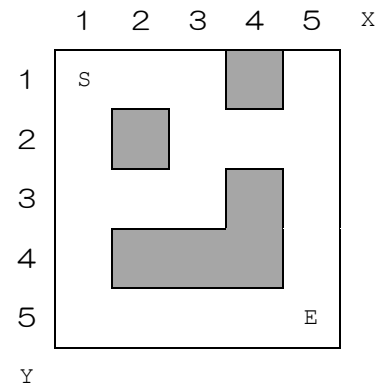


図3-1 迷路

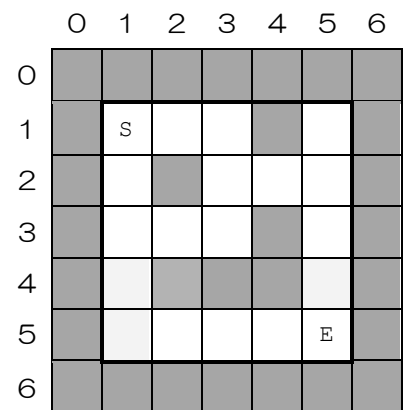


図3-2 外枠付き迷路

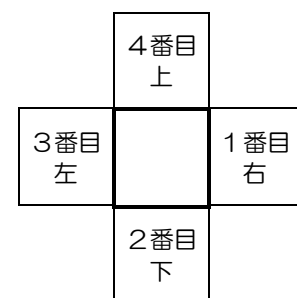


図3-3 探る順

c言語の2次元配列では、横方向をx、縦方向をyとすると、maze[y][x]となる。見た目に、添え字が、x、yという順序ではなく、y、xという順序になることに注意すること。また、配列は大きさを宣言するので、宣言ではmaze[7][7]となり、添え字の範囲は0～6となる。



入口 `maze[1][1]` から出口 `maze[5][5]` に達する道をたどる関数 `findgoal(x,y)` は次のように定義できる。関数 `findgoal(x,y)` は、`maze[y][x]` がゴールであるかどうかを調べる関数である。初期値として、`x=1, y=1` として入口を与えることにする。

- (1) `maze[y][x]` が “`■`” でないならばリターンする。
- (2) `maze[y][x]` が出口ならば “Goal in!!” と表示してリターンする。
- (3) `maze[y][x]` が出口でなければ、次のように4方向を調べる。
  - 右がゴールかどうかを調べる。すなわち、`findgoal(x+1,y)` を実行する。
  - 下がゴールかどうかを調べる。すなわち、`findgoal(x,y+1)` を実行する。
  - 左がゴールかどうかを調べる。すなわち、`findgoal(x-1,y)` を実行する。
  - 上がゴールかどうかを調べる。すなわち、`findgoal(x,y-1)` を実行する。
- (4) 4方向すべてを調べたので、リターンする。

このように定義が再帰的なので、関数 `findgoal(x,y)` も自然と再帰関数となる。このような考えで作成したプログラムが `3-3.c` である。プログラム `3-3.c` をコンパイルして、実行すると図3-4のような出力が得られる。壁が “`■`” であり、経路が “`○`” である。

プログラム `3-3.c` では、“`■`” を2, “`□`” を0, すでに通ったことを示すために1を使っている。それぞれマクロで、`KABE`, `AKI`, `KITA` と定義 (`#define`) してある。また、迷路の定義は、ファイル `maze1` であり、「`#include "maze1.dat"`」とすることによって読み込んでいる。インクルードするファイル（迷路の定義）を変更することにより、いろんな迷路を試すことができる。

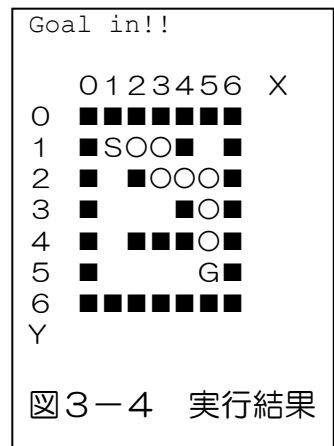


図3-4 実行結果

**課題21** プログラム `3-3.c` の `findgoal()` を変更することによって、入口から出口に至るまでの経路の長さを示すように書き換えよ。書き換えたプログラムは `3-4.c` とせよ。例えば、 $(1, 1) \rightarrow (2, 1) \rightarrow (3, 1) \rightarrow (3, 2) \rightarrow (4, 2) \rightarrow (5, 2) \rightarrow (5, 3) \rightarrow (5, 4) \rightarrow (5, 5)$  の経路の長さは9である。なお、`printmaze()` は書き換えてはいけない。迷路の定義を変更することによって、プログラムの正しさを確認せよ。

**課題22** `findgoal()` が正しく動くことを確認するためのテストデータを考えよ。たとえば、袋小路だけで出口のない迷路、全部ふさがっている迷路、全部空いている迷路などを考えよ。テストする迷路の定義は、`maze1.dat` や `maze2.dat` のように別ファイルとすること。課題を提出するときは、迷路の定義を印刷し、その余白にそのような迷路で試した理由を書け。

**課題23** プログラム `3-3.c` の `findgoal()` を、入口から出口まで至るすべての経路を求めるように変更せよ。書き換えたプログラムを `3-5.c` とせよ。訂正は、1行削除、1行追加だけである。書き換えたプログラム `4-3.c` の動作確認は、課題22で定義した迷路が利用できる。

## 第4回 プログラミング言語演習Ⅱ 迷路をたどる

### ■準備

/home/info/EX/PRO2/4/を自分のホームディレクトリにコピーせよ。

### ■中間試験の範囲について

第4回までの課題及び配布資料に関連した試験を予定しているので、各自十分理解しておくこと。  
また、課題27の内容はレポート課題とする予定である。

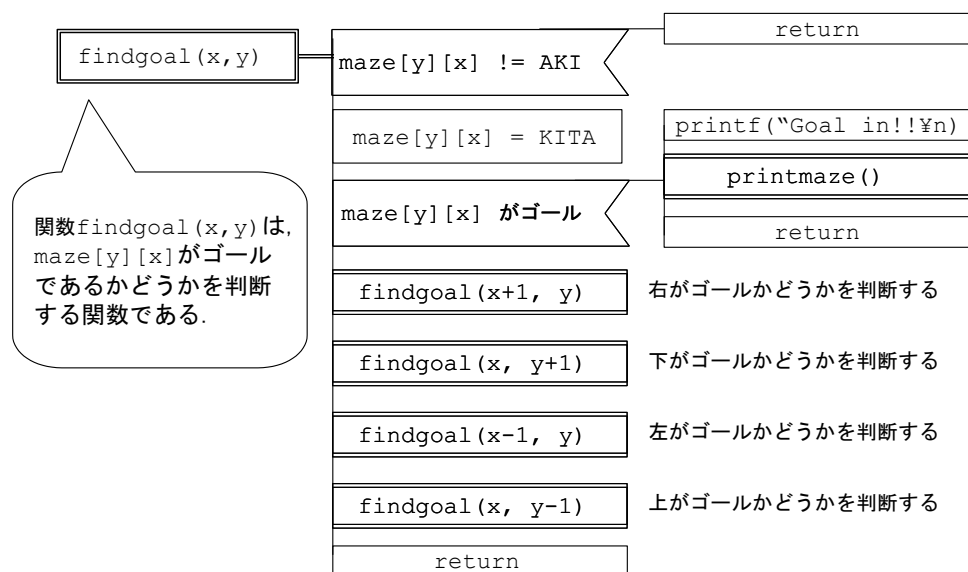
### ■提出物

第3回までの遅れを取り戻すようにすること。分からなければTAに質問すること。

課題24(4-1.c)は必須。できれば25(動作の図示)も提出すること。時間が余った場合は課題26(テストデータ)を行え。課題27は、難しいのですぐに提出することは要求しないが、考えておくこと。

### ■第3回の要点「再帰関数」

前回の関数 `findgoal()` のアルゴリズムを PAD で書くと、次のようになる。(関数 `findgoal()` の呼出しを強調するように書いてある。) この再帰アルゴリズムのポイントは、「関数 `findgoal(x, y)` は `maze[y][x]` がゴールであるかどうかを判定するだけの関数である」と理解することである。



第2～3回で、再帰関数（再帰呼出し）を取り上げた。再帰関数の形は次のようになっている。

```
recursive(引数1, 引数2, 引数3, ...)  
{  
    if( 終了条件の判定 )  
        明らかな解（関数値）を返す;  
    else  
        より小さくなった問題を解くために recursive(引数1, 引数2, 引数3, ...) を呼び出す;  
}
```

終了条件の判定は、関数 `fib()` では `n=0` あるいは `n=1` であった。else 文では、より小さくなった問題を解くために自分自身を呼び出す。すなわち、関数 `fib(n)` では、より小さくなった問題

は  $\text{fib}(n-1)$  と  $\text{fib}(n-2)$  である。このように、再帰関数は、(再帰) 呼出しごとに解くべき問題が小さくなり、最終的には終了条件が成立することになる。これまで扱った再帰関数を調べて、確認せよ。 関数  $\text{findgoal}(x, y)$  の終了条件は何か (課題 25)。

階乗、フィボナッチ数列、組合せなどの再帰関数は、再帰呼出しを使わず `for` ループや `while` ループのプログラムに簡単に書き換えることができる。しかし、迷路をたどるプログラムは、再帰関数を使えば簡単にプログラムできるが、再帰関数を使わないプログラムに書き換えることは難しい (課題 27)。

再帰関数を使うとプログラムは簡単になるが、実行時間は長くなる場合がある。課題 20 では、関数  $\text{fib}()$  と関数  $\text{fib2}()$  の呼出回数を片対数でグラフ化した。このグラフで関数  $\text{fib}()$  の呼出回数は、直線になったはずである。片対数グラフで直線になるということは、 $c$  を定数とすれば呼出し回数は  $c^n$  に比例するということを意味する。( $y=c^x$  の両辺の自然対数を取ると、 $\log(y)=x \times \log(c)$  となる。ここで、 $\log(c)$  は定数になるので、片対数グラフでは直線となる。)

グラフから定数  $c$  を求めると、およそ 1.6 になる。

$$c \cong \exp\left(\frac{\ln(\text{fib}(20)\text{の呼出し回数}) - \ln(\text{fib}(10)\text{の呼出し回数})}{20 - 10}\right) = \exp\left(\frac{\ln(21891) - \ln(177)}{10}\right) = 1.6189$$

すなわち、関数  $\text{fib}()$  の実行時間は、ほぼ  $1.6^n$  に比例する。アルゴリズムの用語でいうと、 $O(1.6^n)$  である (オーダー  $1.6^n$  と読む。)。このようにグラフ化することによって、関数  $\text{fib}()$  の振る舞いを視覚的に把握することができる。課題 13 で作成した関数  $\text{comb}()$  (プログラム 2-2.c) のオーダーはいくつになるだろうか? 迷路のプログラムの場合は、どのようにしてグラフ化すればよいかを考えてみよ。課題として提出してもよい。

**課題 24** 迷路をたどる関数  $\text{findgoal}()$  の動作を調べるためにプログラム 4-1 を作成した。配布したのは実行形式だけである。このプログラムと同じ動作するプログラム **4-1.c** を作成せよ。なお、迷路の定義は、プログラム 3-3.c のように `include` するようなプログラムとすること。

\$4-4 | more ↓

←表示が流れないように more を使って見る。

more ではキャリッジリターンを入力すると 1 行だけ送られる。

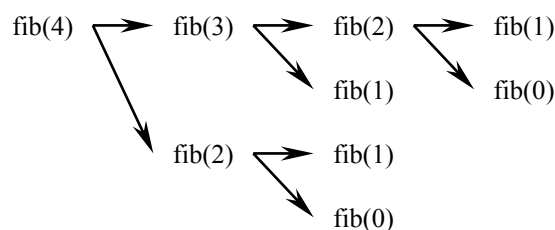
**課題 25** (解答は、プログラム 4-1.c の余白に記入せよ。) 迷路をたどる関数  $\text{findgoal}()$  は、呼び出す毎にどれくらい問題のサイズが小さくなってゆくか? また、迷路をたどる関数  $\text{findgoal}()$  の終了条件は何か?

**課題 26** 課題 24 で作成したプログラム 4-1.c を使って、関数  $\text{findgoal}()$  がどのように再帰呼び出しされているかを調べて、その動作を探索木として図示せよ。関数  $\text{findgoal}()$  の再帰動作を理解するために、いくつかの迷路で試してみよ。どのような迷路 (テストデータ) で試して見れば良いだろうか。テストデータを示すと共に、そのテストデータで何をテストするのかを解説せよ。課題 22 のテストデータと、本課題のテストデータは、テストする内容が異なる。

**課題 27** 関数  $\text{findgoal}()$  は再帰関数である。再帰関数でない  $\text{findgoal}()$  を作成する方法を考えよ。アルゴリズムが思いついたら PAD で図示してみよ。この課題は、レポートと課題となる。

## 揭示資料 第3回の補足

この関数  $\text{fib}()$  の時間計算量が  $O(1.6^n)$  になることを簡単に解説する。この関数の時間計算量は  $\text{fib}()$  の呼出し回数に比例すると考えられる。呼出し回数を、 $\text{fib}(0)$  と  $\text{fib}(1)$  は 1 回、 $\text{fib}(2)$  は 3 回、 $\text{fib}(3)$  は 5 回と数えることにする。 $\text{fib}(4)$  の呼出し回数は、下図に示すように 9 回である。この呼出し回数を  $n$  の式で表すことを試みる。



引数  $n$  のとき、 $\text{fib}(n)$  の計算に必要な呼出し回数を  $C(n)$  とすると、次のように定義できる。これを使えば、 $C(2)=3$ 、 $C(3)=5$ 、 $C(4)=9$  と正しく求まることがわかる。

$$C(n) = C(n-1) + C(n-2) + 1, C(0)=1, C(1)=1$$

さらに、この  $C(n)$  を、フィボナッチ数列  $\text{fib}_n$  を使って<sup>1</sup>、次のように表すことができる<sup>2</sup>。

$$C(n) = 2 \cdot \text{fib}_n - 1$$

すると、 $\text{fib}(n)$  の時間計算量は、次式で示すように  $O(1.618^n)$  と見積もることができる<sup>3</sup>。

$$\begin{aligned} O(C(n)) &= O(2 \cdot \text{fib}_n - 1) = O\left(2 \cdot \left\{ \left( \frac{1+\sqrt{5}}{2} \right)^{n+1} - \left( \frac{1-\sqrt{5}}{2} \right)^{n+1} \right\} \cdot \frac{1}{\sqrt{5}} - 1\right) \cong O\left( \left( \frac{1+\sqrt{5}}{2} \right)^{n+1} \cdot \frac{1}{\sqrt{5}} \right) \\ &\cong O\left( \frac{1.618^{n+1}}{\sqrt{5}} \right) \cong O(1.618^n) \end{aligned}$$

以上

<sup>1</sup>  $C(n)$  を表すのにフィボナッチ数列を使うことができる。

<sup>2</sup>  $C(n) = 2 \cdot \text{fib}_n - 1$  が成り立つこと証明は帰納法を使えばよい。フィボナッチ数列  $\text{fib}$  の一般項を求める方法は高校レベル。

<sup>3</sup> この 1.618 という値は黄金比に表れる値でもある。

---

## 第5回 プログラミング言語演習Ⅱ スコープ、ポインタ、スタック

---

### ■お知らせ

次回（11月7日）は中間試験を実施する。教室は、J1ではなく大学院棟409講義室に変更する。範囲は第1回～第5回（課題33まで）である。教科書だけ持込可とする。出席や課題提出は、進捗状況のチェックの意味合いが大きい。合格するためにも、中間試験で良い点数を取ること。

### ■準備

/home/info/EX/PRO2/5/を自分のホームディレクトリにコピーせよ。

### ■今回の内容について

前回の課題が終わっていない者は、前回の課題を進めて、今回の課題は宿題とする。試験範囲は、課題33までなので、今週中に勉強しておくこと。

今回の課題は、教科書の「5.4 さまざまな変数」（189頁）、「5.5 変数の通用範囲（スコープ）」（199頁）、「8.1 ポインタの基礎」（270頁）の内容を扱う。提出物は少ないが、スコープ、アドレス演算子、ポインタについて、教科書を見ながら課題を進めること。

### ■課題と提出物について

課題28～31については、提出物はない。教科書を見ながら、スコープについて理解せよ。

課題32～33については、修正したプログラム5-5.cと、その余白に課題32を記入せよ。

課題34～35については、提出物はない。配布したプログラムを十分理解せよ。

課題36については、作成したプログラム5-9.cを提出せよ。

変数のスコープ（199頁）とアドレス演算子“&”
--------------------------

課題20で関数の呼出し回数を数えるプログラムを作成した。教科書（194頁）のように、多くの人は外部変数を使ったと思う。外部変数は、どの関数からも見えるので、簡単に使える。しかし、どの関数で操作されているかが分かり難いので、外部変数は好んでは使われない。

**課題28** プログラム5-1.cをコンパイルして、出力されるエラーを確認せよ。関数main()中で宣言されている自動変数kは関数fact()からは見えない。変数kのスコープは関数main()の中だけということである。

**課題29** プログラム5-1a.cと5-1b.cをコンパイルすると、エラーは出力されない。しかし、エラーは出力されないが、意図通りのプログラムでない。3つのプログラム（5-1.c, 5-1a.c, 5-1b.c）を例として、自動変数とそのスコープ（通用範囲）について解説せよ。（関連 課題31）

**課題30** アドレス演算子“&”（273頁）を使うと、変数が配置されているメモリ中のアドレスを知ることができる。プログラム5-2.cは、外部変数kのアドレスを関数main()と関数fact()で表示している。出力されるアドレスが同じということは、同じ変数が見えているということである。すなわち、外部変数のスコープはファイル内であることがわかる。プログラム5-2.cを実行することによって確認せよ。

**課題31** プログラム 5-3.c では、外部変数 `k` と関数 `main()` の自動変数 `k` が使われている。変数名は同じ `k` であるが、スコープが異なる。関数 `fact()` から見える変数 `k` は、外部変数である。2つの変数 `k` のアドレスを出力することによって、名前は同じであるが、スコープが異なる変数であることがわかる。プログラム 5-3.c を実行することによって、このことを確認せよ。

### ポインタ

課題31で、関数 `main()` の自動変数 `k` を関数 `fact()` で参照するには、関数 `main()` の自動変数 `k` のアドレスを関数 `fact()` に伝えればよい。変数の値ではなく“変数のアドレス”を扱うには、ポインタ変数を使う。ポインタ変数は“`int *k`”のように、型 (`int`) に“\*”をつける。すなわち、ポインタ変数を使えば、変数のアドレスを介して、変数を操作できるようになる。

**課題32** プログラム 5-4.c では、関数 `main()` は“`&k`”として自動変数 `k` のアドレスを関数 `fact()` に渡している。一方、関数 `fact()` は“`int *k`”のようにポインタ変数として受け取っている。このため関数 `fact()` 内の `printf` 文を見れば分かるように、“`&k`”ではなく、“`k`”でアドレスが正しく表示できている。すなわち、関数 `main()` の自動変数 `k` のアドレスが関数 `fact()` のポインタ変数 `*k` に格納されている。ポインタ変数に格納されている変数のアドレスを介して、その変数の値を参照するためには、“`*k=*k+1`”のようにポインタ変数に“\*”を付加すればよい。プログラム 5-3.c と 5-4.c を比較して、ポインタ変数の使い方を理解せよ。ポインタ変数を使うことによって、スコープはどのように変わるかを解説せよ。

**課題33** プログラム 5-5.c は、関数 `fib()` を呼出し回数を数えるプログラムであるが、正しく動作しない。課題32と同じような考え方で、関数 `main()` の自動変数 `k` のアドレスを関数 `fib()` に渡して、プログラムが正しく動作するように 5-5.c を修正せよ。

## ■スタック (stack) とは

スタックについての詳しい解説は、「データ構造とアルゴリズム」の講義や教科書で扱っているのでそちらを参照すること。ここでは簡単なスタックを次に示す。スタックは基本的なデータ構造の1つであり、簡単なスタックは、1次元配列 `stack[]` と、スタックポインタと呼ぶ1つの変数 `sp` で実現できる。スタックにデータを1つ追加する操作をプッシュダウン (push down, 略して push), スタックからデータを取り出す操作をポップアップ (pop up, 略して pop) と呼び、関数 `push()` と関数 `pop()` は、2つの操作に対応する関数である。その他に、スタックを初期化 (`sp` を初期化する) する操作として関数 `initstack()` を用意した。

## ■スタック操作の関数についての解説

左に示す3つの関数はスタック操作のための関数である。関数 `push()` と `pop()` は、それぞれ引数をプッシュダウンとポップアップする関数である。正常に終了した場合1 (真) を返し、そうでない場合0 (偽) を返す。このような仕様にしておくこと、

`if( !push(x) ) エラー処理`  
と書くことができる。“!” は否定演算子であり、この例では、「`push()` できなければエラー処理」とプログラムを読むことができる。

関数 `initstack()` は、スタックを初期化する関数である。この関数は常に成功するので関数値は常に1である。さて、`push()` と `pop()` が関数として定義されていることは理解できるが、単に `sp` を操作しているだけの関数 `initstack()` は

```
int initstack()
{
    sp = 0 ;
    return 1;
}

int push( int d )
{
    if( sp == STACKSIZE ) return 0;
    stack[sp]=d;
    sp = sp + 1;
    return 1;
}

int pop( int *d)
{
    if( sp==0 ) return 0 ;
    sp = sp - 1 ;
    *d = stack[sp];
    return 1;
}
```

どうして必要なのだろうか？ スタックを使う側としては、変数 `sp` の値が知りたいのではなく、「スタックを初期化したい」ということが実行できればよいのである。例えば、関数 `initstack()` では `sp` を 0 に初期化しているが、`STACKSIZE-1` に初期する実現方法も考えられる。この場合、関数 `push()` では `sp=sp-1`、関数 `pop()` では `sp=sp+1` という操作をすることになる。プログラムを開発していると、より効率のよい方法が見つければ、それに合わせてプログラム (関数) を変更したい。そのようなとき、スタックを初期化するのに “`sp==0;`” のようなプログラムを書いてしまうと、プログラム (関数) を変更することが難しくなる。例えば、スタックポインタの変数名が `sp` から `longsp` に変更されたとする。すると、プログラム中で表れる `sp=0` をすべて `longsp=0` に変更しなくてはならない。しかし、関数 `initstack()` を使っていれば、変数の変更箇所はスタックを操作する関数だけに限定される。このように関数の使い方の部分 (インタフェース) とプログラム (実装) を分離して考えることによって、プログラムしやすくなるのである。

**課題34** プログラム 5-6.c を実行し、関数 `push()`、`pop()` の動作を理解せよ。STACKSIZE の値を 10 に変更したときどうなるか？ 関数 `pop()` の引数が “`int *d`” であるのは何故か？ またアルファベットが逆順に出力されるのは何故か？

プログラム 5-6.c には、関数 `main()` だけでなくスタック操作の関数 `push()` や `pop()` などが含まれている。関数 `push()` や `pop()` の使い方と実装を分離するために、分割コンパイルやヘッダファイルが利用される。ちょうど、三角関数 `sin()` を利用するには、“`#include <math.h>`” とコンパイル時にリンク “`-lm`” とするのと同じである。

**課題35** プログラム 5-7.c は、プログラム 5-6.c と同じ動作をする。しかし、変数のスコープを利用して、関数 `pop()` や `push()` などの実装（中身）が分からないようになっている。また、スタックポインタである変数 `sp` は、関数 `main()` からは見えない。プログラム 5-7.c をコンパイルするには、次のような操作が必要である。ヘッダファイル `stack.h` を見て “`extern`” と “`static`” の使い方を理解せよ。

```
cc -c stack.c          ←この操作により、stack.o が生成される。
cc -o 5-7 5-7.c stack.o ←stack.o をリンクして、a.out が生成される。
```

上記の操作で、`stack.o` は一度だけ生成すればよい。もし、他のプログラムでスタックを使いたい場合は、“`#include "stack.h"`” とすれば関数 `pop()` などが使える。そして、コンパイルするときに `stack.o` をリンクするには、“`cc XXX.c stack.o`” とすればよい。

**課題36** 実行形式 5-8 を配布した。このプログラム 5-8 を実行すると、スタートからゴールまでの道のりが数値として出力される。迷路を出力する関数 `printmaze()` は、ファイル 5-8printmaze.c として配布してある。実行形式 5-8 と同じ動作するプログラム 5-9.c を作成せよ。ただし、迷路を出力する関数は 5-8printmaze.c を使うこと。

5-8printmaze.c は次の通り。

```
1: void printmaze(void)
2: {
3:     int x,y;
4:     printf("¥n");
5:     printf("    0 1 2 3 4 5 6 X ¥n");
6:     for( y = 0 ; y < MAZE_Y ; y ++ )
7:     {
8:         printf("%2d ",y);
9:         for(x = 0 ; x < MAZE_X ; x++ )
10:        {
11:            if( maze[y][x] == KABE ) printf("■");
12:            else if( maze[y][x] == KITA ) printf("%2d",trace[y][x]);
13:            else printf(" ");
14:        }
15:        printf("¥n");
16:    }
17:    printf(" Y¥n");
18:    printf("¥n");
19: }
```



## 第6回 中間試験（範囲：第1回～第5回）

※解答用紙が不足した場合は申し出ること。

問題はどの順番で答えても構わないが、回答と問題の対応が明確につくように答えること。

**課題34** 右に示す2つの関数 `fact1()` と `fact2()` は、共に階乗を求める関数である。これら2つの関数に関するテストデータを示せ。テストデータの形式は、次のようにすること。欄「テストの目的」には、そのテストデータで何をテストするのかを記入せよ。

引数	正しい関数値	<code>fact1()</code> の関数値	<code>fact2()</code> の関数値	テストの目的

```
int fact1( int n )
{
    int v = 1 ;
    do {
        v = v * n ;
        n = n - 1 ;
    } while( n != 0 );
    return v ;
}

int fact2( int n )
{
    if( n == 0 ) return 1 ;
    return n * fact2( n - 1 );
}
```

**課題35** 乗算  $a*b$  は、次式のように再帰式として表すことができる。両辺に乗算演算子 “\*” が含まれているので、再帰関数として定義できる。関数名を  $m(a, b)$  とし、プログラムを示せ。

$$a * b = a + a * (b - 1) \quad \text{ただし, } b > 0 \text{ とする.}$$

**課題36** ホーナー法は、多項式を効率よく計算する方法である。一般に、 $a_n, a_{n-1}, \dots, a_1, a_0$  を係数とする  $n$  次の多項式  $f(x)$  をホーナー法で直すと、次のようになる。

$$\begin{aligned} f(x) &= a_0x^n + a_1x^{n-1} + \dots + a_{n-1}x + a_n = x(a_0x^{n-1} + a_1x^{n-2} + \dots + a_{n-1}) + a_n \\ &= \dots \\ &= x(x \dots (x(a_0x + a_1) + a_2) + \dots + a_{n-1}) + a_n \end{aligned}$$

ここで  $f(x) = p_n$  と置くと、 $p_n$  は  $p_0 = a_0, p_k = p_{k-1}x + a_k (k=2, 3, \dots, n)$  となり、再帰式として表すことができる。 $p_n$  を再帰関数として定義できる。関数名を  $h(\text{float } x)$  とし、プログラムを示せ。ただし、次数  $n$  は「`#define N 10`」のように与えられ、係数  $a_n$  は外部変数として配列  $a[N]$  として与えられるとせよ。

**課題37** 組合せの数を計算する関数  $c1(n, k)$  と  $c2(n, k)$  について考える。関数  $c1()$  は、再帰関数であり、式  ${}_nC_k = {}_{n-1}C_{k-1} + {}_{n-1}C_k, {}_nC_0 = {}_nC_n = 1$  に基づく。一方、関数  $c2()$  は、式  ${}_nC_k = \frac{n(n-1)(n-2)\dots(n-k+1)}{1 \cdot 2 \cdot 3 \dots k}$  に基づく。関数  $c1()$  と  $c2()$  のプログラムを示せ。

**課題38** 課題37の関数  $c1()$  の呼び出し回数を調べたい。関数 `main()` 内の自動変数として宣言された `int` 型変数 `t` に関数  $c1()$  の呼び出し回数が記録されるように関数  $c1()$  を修正したプログラムを示せ。

**課題39** 課題37の関数  $c1()$  と  $c2()$  の実行効率を比較したい。関数  $c1()$  については、課題38のように呼び出し回数を使う。それでは、関数  $c2()$  については、どのような値を使えばよいかを考えよ。そ

して、その値を記録するように関数 `c2()` を修正したプログラムを示せ。

**課題40** 次のプログラムは、入口から出口まで至るすべての経路を出力するプログラムの一部である。課題23では、30行目の `return` 文の削除と、37行目の “`maze[y][x]=AKI;`” の挿入が答えであった。このプログラムが、どうしてすべての経路を求めることができるのかを詳細に解説せよ。必要ならば探索木などの図を使っても良い。

```
1  #include <stdio.h>
2  #define MAZE_X 7
3  #define MAZE_Y 7
4  #define AKI 0
5  #define KITA 1
6  #define KABE 2
7  #define START_X 1
8  #define START_Y 1
9  #define GOAL_X 5
10 #define GOAL_Y 5
11
12 #include "maze1.dat"
13
14 void printmaze(void);
15 int findgoal(int, int);
16
17 int main(void)
18 {
19     findgoal(START_X, START_Y);
20 }
21
22 int findgoal(int x, int y)
23 {
24     if( maze[y][x] != AKI ) return ;
25     maze[y][x] = KITA;
26     if( x == GOAL_X && y == GOAL_Y )
27     {
28         printf("Goal in!!\n");
29         printmaze();
30         //      return ;
31     }
32
33     findgoal(x+1, y);
34     findgoal(x, y+1);
35     findgoal(x-1, y);
36     findgoal(x, y-1);
37     maze[y][x] = AKI;
38     return ;
39 }
```

**課題41** 課題40のプログラムで、35行目が `findgoal(y-1, y)` であるとき、このプログラムはどのように動作するかを簡単に述べよ。そして、このバグを発見できるテストデータと発見できないテストデータを示せ。

**課題42** 課題41の解答から、迷路のプログラムのテストデータはどのように作成すればよいかを考察せよ。

**問題A** （時間が余った人は答えてください。）プログラミング言語・演習Ⅱでは、プログラミング言語・演習Ⅰで学んだC言語の基礎的な知識を想定している。しかし、課題の提出状況はあまりよくない。プログラミング言語演習に対する要望や意見があれば、自由に書いてください。

## 第7回 プログラミング言語・演習Ⅱ 迷路プログラムとデータ構造

### ■準備

/home/info/EX/PRO2/7 をコピーすること。また、/home/info/EX/PRO2/6 も内容が更新されているのでコピーしておくこと。

### ■スタックを使って迷路をたどる

第6回の課題に、関数 `push()` と `pop()` を使って、再帰関数 `findgoal()` の動作を調べるプログラム `list1.c` を示した。`list1.c` を動かして、スタックをどのように使えば良いかを検討せよ。

再帰を使わないで迷路をたどるアルゴリズムの概略は、スタックを使うと図1のように表せる。

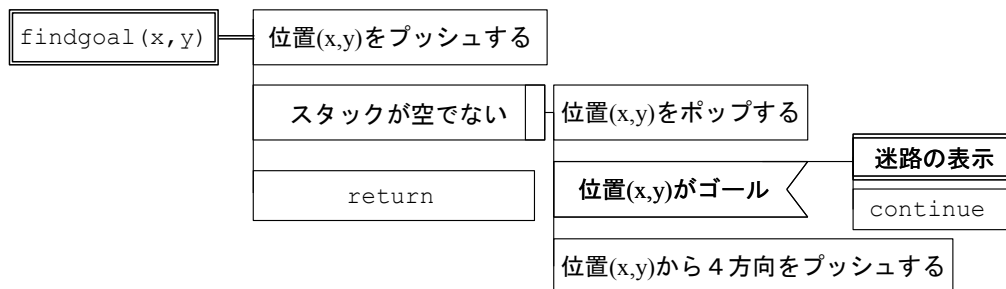


図1 再帰でない迷路をたどる大まかなアルゴリズム

このアルゴリズムを元にして、プログラム 7-1.c を作成した。このプログラムの動作を図2に示す。図中、ゴールの (3,3) が3回出現しているので、3通りの経路をたどったことがわかる。



図2 スタックの変化

ところで、図2で扱った3×3の迷路で、S から E までの最短の経路を調べるにはどうしたらよいだろうか？ 実は簡単で、配列 `dist[][]` を用意し、この配列にはスタートからの距離を格納する。すなわち、`dist[1][1]=1` であり、(1,1) から長さ1でたどれる範囲には2を代入し、長さ3でたどれる範囲には3を代入するということに、ゴールまで続けてゆく。すると、図3に示すように、ゴールでは `dist[3][3]=5` となり、最短経路が5であることがわかる。

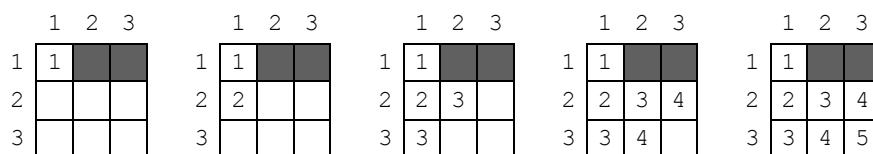
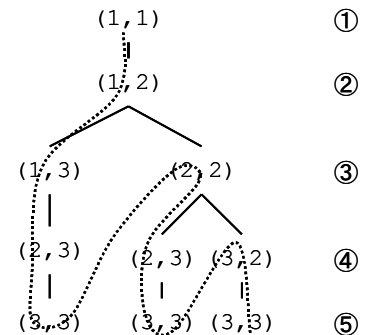


図3 最短経路の求め方

**課題39** プログラム 7-1.c は、図3で示した考え方を取り入れたプログラムである。すなわち、4方向の空きを調べて、その位置をプッシュするとき、距離が最短でない位置はプッシュしないようにする。プログラム 7-1.c の60行目を埋めて、プログラムを完成させ、動作を確認せよ。プログラム 7-1.c を提出するとき、余白に、関数 main() の29～31行目で配列 dist[][] を大きな整数値で初期化しているが、その理由を書け。

さて、スタックを使ったプログラム 7-1.c が迷路をたどる様子を右図に示す。この図を探索木とよんだりする。①から⑤は、木の深さを表しており、配列 dist[][] に格納される値と同じである。スタート (1, 1) からゴール (3, 3) までの3通りの経路があることがわかり、破線がたどる順序を示している。この破線のたどり方を「深さ優先探索」あるいは「縦型探索」という。迷路では、ゴールまでの経路を1つずつ探すことに相当する。



#### ■待ち行列を使って迷路をたどる

待ち行列 (queue, キュー) は、スタックと対比されるデータ構造である。スタックは LIFO (Last In First Out, 最後に入れたデータを最初に取り出す) であるが、キューは FIFO (First In First Out, 最初に入れたデータを最初に取り出す) である。基本操作は、キューに入れる enqueue と、キューから取り出す dequeue である。キューを配列で実現する場合、図4に示すように、2つの変数 front と rear を使う。変数 rear はデータを入れるとき使い、変数 front はデータを取り出すとき使う。

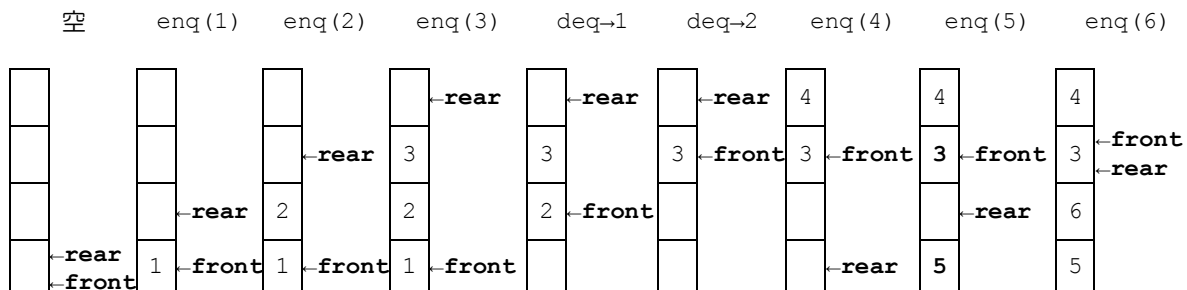


図4 1次元のキューの操作 (enq は enqueue, deq は dequeue である)

配布したプログラム queue2.c では、関数 enqueue(int, int) と関数 dequeue(int\*, int\*) の2つの基本操作の関数が含まれている。プログラム 7-1.c で、関数 push() を関数 enqueue() に置き換え、関数 pop() を関数 dequeue() に置き換えても、プログラムは正しく迷路をたどることができる。

**課題40** プログラム 7-1.c の関数 push() と pop() を関数 enqueue() と dequeue() に置き換えたプログラム 7-2.c を作成し、動作を確認せよ。関数 printstack() は関数 printqueue() に置き換えよ。配布した queue2.c は関数 dequeue() が未完成なので完成させて queue2.c として提出せよ。プログラム 7-2.c の動作を図5に示す。



Figure 1: A diagram illustrating the evolution of a system over time steps 1 to 5. The diagram shows a branching structure with nodes labeled by coordinates  $(x, y)$ . Solid lines represent the main evolution, while dotted lines represent a specific path. The nodes are:  $(1, 1)$  at step 1;  $(1, 2)$  at step 2;  $(1, 3)$  and  $(2, 2)$  at step 3;  $(2, 3)$ ,  $(2, 3)$ , and  $(3, 2)$  at step 4; and  $(3, 3)$ ,  $(3, 3)$ , and  $(3, 3)$  at step 5. The path starts at  $(1, 1)$ , goes to  $(1, 2)$ , then to  $(1, 3)$ , then to  $(2, 3)$ , then to  $(3, 3)$ , and finally to  $(3, 3)$ .

データ構造とアルゴリズムという視点でみると、スタックを使えば深さ優先探索になり、キューを使えば幅優先探索になることは面白い。プログラムでは、関数 `push()` と関数 `pop()` をそれぞれ関数 `enqueue()` と関数 `dequeue()` に置き換えるだけである。幅優先探索と深さ優先探索のどちらかを使うかは、扱う題材や期待する探索結果に応じて決める必要がある。迷路をたどるときは、スタートからゴールまでの経路も知りたいので、深さ優先探索を選択することになる。これまで扱った再帰関数 `findgoal()` は、暗黙的にスタックを使っているの、深さ優先探索となっている。再帰関数 `findgoal()` を幅優先探索に変更することは可能だろうか？

## レポート課題のヒント

課題では、すべての経路を見つける必要がある。すなわち、次のような探索木を深さ優先で探索すればよい。今回の課題のように配列 `dist[][]` は使うことができない。プログラムの基本的な構造は、深さ優先探索と同じになる。しかし、図1で示したアルゴリズムのうち、「位置(x,y)から4方向をプッシュする」という処理を工夫する必要がある。ひとつの方法として、スタックのデータ構造を  $(x, y, d)$  とし、 $d$  は探索方向を示すことにする。すなわち、ポップしたとき、 $d$  の値を調べて、次に探索すべき方向を決定すればよい。このようにスタックのデータ構造を少し変えて作成したプログラムが配布したプログラム `KadaiRef` である。

プログラム `KadaiRef` は、関数 `push(x, y, d)` と関数 `pop(&x, &y, &d)` を使っている。3×3の迷路の実行結果と探索木を次に示す。実行結果で、`POP(3, 3, 0)` の下に表示されているのが、スタックの中身である。これを見ると、 $(1, 1, \text{下}) \rightarrow (1, 2, \text{右}) \rightarrow (2, 2, \text{右}) \rightarrow (3, 2, \text{下}) \rightarrow \text{ゴール}$  と探索していることが分かる。

```
Goal in!!
POP(3, 3, 0)
(1, 1, 2), (1, 2, 1), (2, 2, 1), (3, 2, 2)
```

```
  0 1 2 3 4 X
0 ■■■■■■
1 ■ 1■■■■■
2 ■ 2 3 4■
3 ■    5■
4 ■■■■■■
Y
```

```
Goal in!!
POP(3, 3, 0)
(1, 1, 2), (1, 2, 1), (2, 2, 2), (2, 3, 1)
```

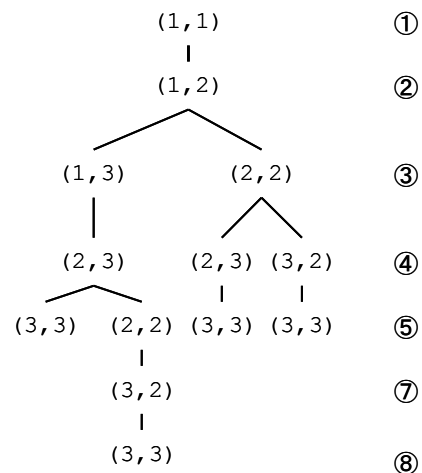
```
  0 1 2 3 4 X
0 ■■■■■■
1 ■ 1■■■■■
2 ■ 2 3  ■
3 ■    4 5■
4 ■■■■■■
Y
```

```
Goal in!!
POP(3, 3, 0)
(1, 1, 2), (1, 2, 2), (1, 3, 1), (2, 3, 1)
```

```
  0 1 2 3 4 X
0 ■■■■■■
1 ■ 1■■■■■
2 ■ 2    ■
3 ■ 3 4 5■
4 ■■■■■■
Y
```

```
Goal in!!
POP(3, 3, 0)
(1, 1, 2), (1, 2, 2), (1, 3, 1), (2, 3, 4), (2, 2, 1), (3, 2, 2)
```

```
  0 1 2 3 4 X
0 ■■■■■■
1 ■ 1■■■■■
2 ■ 2 5 6■
3 ■ 3 4 7■
4 ■■■■■■
Y
```



# プログラミング言語演習Ⅱ

担当 富澤

レポート提出記録	判定・指示	
提出期限年月日		
平成 年 月 日		
再提出期限年月日		
平成 年 月 日		
課題		
<p>これまでの課題は、小さなプログラムで、実際に実行してみれば自分自身で正しいかどうか判断できる。最後の課題は、これまでの課題の集大成であり、自分のプログラムやアルゴリズムを正しく読み手に伝えるということを重視する。レポート課題は次のとおり。裏面の「レポートの書き方」にしたがってレポートを書くこと。</p> <p>（課題）再帰呼出しを使わないで、迷路をとくプログラムを作成せよ。すべての道を見つけること（全解探索）。これまでの課題を参考にし、スタック（stack2.c, stack2.h）を使うように関数findgoal()を修正せよ。また、ゴールまで経路を表示すること。配布したKadaiRefの実行結果を参考にせよ。</p> <p>情報棟の改修工事のためレポートボックスの設置場所が不明である。このため提出先は、後で連絡する。提出期限までに時間があるが、それだけの時間を必要とする内容を要求している。わからないことがあれば、TAあるいは富澤に質問すること。</p>		
所属コース	学籍番号	氏名

## レポートの書き方

レポートの原則は、自分の考えたことを相手に分かりやすく書く（伝える）ということである。読みにくいと、どんなに内容が優れていても評価は悪くなる。レポート枚数については特に条件はつけないが、自分の理解したことを相手に伝えるのに十分な枚数を書くこと。少ない場合は、書き足りていないことが多い。レポートを書くにあたり次の点に注意すること。

- (1) レポートには、課題に対する解答及び考察、プログラムリスト、テストデータを含めること。  
プログラムリストは、プログラム名と行番号を付けること。プログラムリストやテストデータ及び実行結果などは無駄の無いように印刷すること。レポートはA4判、横書きとする。本用紙を表紙とすること。日本語の正書法に従って書くこと。段落は1字下げ、句読点をはっきり書く。ワープロなどを使用する場合、余白は左20ミリ、右20ミリ、上30ミリ、下30ミリとする。文字ピッチは、10.5ポイントの場合、4.0ミリ以下、行ピッチは文字ピッチの1.6～2.0倍とし、A4判の紙に読みやすく印字する。レポートには必ずページ番号を打つ。場所は、中央下とする。手書きの場合も、これに従うこと。特にレポート用紙で、字間が詰まりすぎているものは読まない。
- (2) 自分の言葉で書くこと。個性ある文章を歓迎する。書籍やホームページなどから借りてきたような文章は書かないこと。引用をする場合には、括弧でくくるなどして範囲を示し、出典を明示すること。引用の明示がないと盗用になる。盗用が発見された場合には即不合格とする。演習のレポートとして、内容のないものは評価の対象としない。
- (3) レポートは、論理的項目に分け、番号と見出しをつける。項目の中は段落に分ける。他人とは違うことを書け。次の項目を適切な構成で番号をつけて含めること。項目名や順番は、各自が書きやすいように修正すること。教科書の書き方は参考になるだろう。
  - ① 課題の目的、内容、説明（配布資料を写すのではなく、自分の言葉で書くこと）
  - ② 計算手順の方針（アルゴリズム）、
  - ③ 変数やデータ構造の説明
  - ④ プログラムの解説
  - ⑤ そのプログラムが正しいことの確認と論証、テストデータの性質、そのプログラムの不十分な点、余力があれば文の実行回数の定量的評価（計算量）や計算時間の定量的評価
  - ⑥ 得られた結果についての解釈、意義、考察、まとめ
  - ⑦ 講義に対する評価、意見、注文、批判、質問、その他自由な項目
- (4) レポートは、自分の考えたことを読み手に正しく伝わるように書くことである。自分の書いたレポートを友人に読んでもらおうと良い。友人がレポートを読んだだけで、アルゴリズムを理解し、そのアルゴリズムをプログラミングできれば、レポートは完成しているといえる。