● 今日の実習

【サンプルプログラム】

★ 再帰の除去

一般に再帰呼び出しを用いるプログラムは再帰を用いない形に書き直すことができる．その手続きは次のようなものである．

```
foo(n) {
 CODE A ;
 if (CONDITION A) {
  CODE B ;
  m = n ;
  foo(m) ;
  CODE C ;
 }
 CODE D ;
}
```

```
foo(n) {
 CODE A ;
 while (CONDITION A) {
  CODE B ;
  PUSH() ;
  m = n ;
  CODE A ;
 }
 CODE D ;
 while (!STACKEMPTY()) {
  POP() ;
  CODE C ;
  if (CONDITION A) {
   CODE D ;
  }
 }
}
```

- 関数を呼び出す際には, その時点でのプログラムの状況（呼び出し側の関数内の局所変数などの値）を保存して,（「スタックに積む」と呼び, **push** と呼ばれる）関数呼び出しを行い, それが終了した際には保存した値を復旧するという手続きをとる．（**pop** と呼ぶ）その際に用いるデータ構造として,「**スタック**」(**stack**) と呼ばれるものを用いる．スタックとは,「先に格納したデータが後から取り出される」という特殊な使い方をするデータ構造である．
- 再帰を除去するためには, 再帰呼び出しを行う時点での状況をスタックに積んでから再帰呼び出しと同じ手続きを実行すればよい．一連の再帰に対応する手続きが終了した後, スタックが空になるまで `POP` を行うことで再帰を除去することができる．
- このように, 一般には再帰を除去するためには「スタックを構成する」ことが必要となる．
- しかし, 末尾再帰の場合には, 一連の再帰呼び出しの終了後に行う手続きが無いため, スタックに保存した情報を再度用いることはない．そのため, 再帰の除去が非常に容易に行うことができる．


```
foo(n) {
 CODE A ;
 if (CONDITION A) {
  CODE B ;
  m = n ;
  foo(m) ;
 }
}
```

```
foo(n) {
 CODE A ;
 while (CONDITION A) {
  CODE B ;
  PUSH() ;
  m = n ;
  CODE A ;
 }
 while (!STACKEMPTY()) {
  POP() ;
 }
}
```

```
foo(n) {
 CODE A ;
 while (CONDITION A) {
  CODE B ;
  m = n ;
  CODE A ;
 }
}
```

すなわち, 末尾再帰のコードに対しては, 上の書き換えによって容易に非再帰なコードに書き換えることができる．そればかりか, スタックを必要としないため, 新しい変数（データ領域）を用意しなくても再帰を除去することが可能である．この操作を「**末尾再帰の除去**」(**elimination of tail recursion**) と呼ぶ．

`ex13-10.c` は前回の `ex13-3.c` の末尾再帰関数 `recursion_tail` を非再帰に書き直したものである．

`ex13-11.c` は前回の `ex13-3.c` の末尾再帰でない関数 `recursion_head` を非再帰に書き直したものである．

なお, 問題によってはアルゴリズムを見直すことにより, 再帰を除去する際にスタックなどを用いる必要のない問題も存在する．

【課題】

★ `exercise-13-10` ２つの正の整数を求めるユークリッドの互除法のプログラムを再帰を用いて書くと, それは末尾再帰となっている．この末尾再帰を除去したコードを書き, それが通常のユークリッドの互除法と同じであることを確かめなさい．

★ `exercise-13-12` （**難？**）ハノイの塔のプログラムの末尾再帰の部分を非再帰に書き直しなさい．

★ `exercise-13-13` （**難**）ハノイの塔のプログラムを非再帰に書き直しなさい．

★ `exercise-13-14` **アッカーマン関数** (**Ackermann function**) とは, 非負整数 m, n に対して整数値をとる以下のように定義された関数である．

$$A(m,n) = \begin{cases} n+1, & \text{if } m = 0 \\ A(m-1,1), & \text{if } n = 0 \\ A(m-1, A(m,n-1)), & \text{otherwise} \end{cases}$$

アッカーマン関数の値を求めるプログラムを書きなさい．その際, 各 $A(m,n)$ を求めるために何回関数を呼び出したかも出力しなさい．

【注意】 アッカーマン関数は「帰納的関数」ではあるが,「原始帰納的関数ではない」関数の例としてよく知られています．

【その他】 電子メールで「今日の講義の感想や意見」を送ってください．

★ ex13-10.c の内容

```
/* $Id: ex13-10.c,v 1.4 2004-07-06 10:50:31+09 naito Exp $ */
#include <stdio.h>
int     recursion_tail(int) ;
int main(int argc, char **argv)
{
  printf("ret=%d\n", recursion_tail(9))   ;
  return 0 ;
}
int recursion_tail(int n)
{
  printf("%d\t", n) ;
  if (n == 0) return 0 ;
  while(n > 0) {
    n -= 1 ;
    printf("%d\t", n) ;
  }
  return n ;
}
```

★ ex13-11.c の内容

```
/* $Id: ex13-11.c,v 1.4 2004-07-06 10:51:27+09 naito Exp $ */
#include <stdio.h>
int     recursion_head(int) ;
int main(int argc, char **argv)
{
  printf("ret=%d\n", recursion_head(9)) ;
  return 0 ;
}
int recursion_head(int n)
{
  int   a[10], index=0, ret_value, m ;
  ret_value = n ;
  m = n ;
  while(index < n) {
    a[index++] = m ;
    m -= 1 ;
  }
  while(index >= 0) printf("%d\t", a[index--]) ;
  return ret_value ;
}
```


● 前回のキーポイント

- 「再帰」または「再帰的関数呼び出し」とは, **関数内部において自分自身の関数を呼び出すこと**である.
- 問題が「帰納的定義」（漸化式）で表現されているときには, 再帰を用いることができる.
- 再帰は, プログラムが問題の帰納的定義そのものを書けばよいので, アルゴリズムの実装が非常に容易になる.
- 再帰を用いるとプログラムの実行速度が低下する.
- 「**線形再帰**」(**linear recursion**) とは, 関数内部で自分自身を**高々１回**のみ呼び出す再帰のことである. 線形再帰は高速に動作するが, 線形でない再帰は速度が低下する.
- 「**末尾再帰**」(**tail recursion**) とは, 関数内部で**再帰的関数呼び出しを行った後に何も操作が無い**再帰のことである.

● 前回の課題の解説

★ exercise-13-1 ２つの正の整数を求めるユークリッドの互除法のプログラムを再帰を用いて書きなさい.

```
/* $Id: exercise13-1-1.c,v 1.2 2004-07-02 11:01:27+09 naito Exp $ */
#include <stdio.h>
int gcd(int, int) ;
int main(int argc, char **argv)
{
  printf("%d\n", gcd(120, 180)) ;
  return 0 ;
}
int gcd(int a, int b)
{
  if (b == 0) return a ;
  return gcd(b,a%b) ;
}
```

```
/* $Id: exercise13-1-2.c,v 1.1 2004-07-02 10:59:07+09 naito Exp $ */
#include <stdio.h>
int gcd(int, int) ;
int main(int argc, char **argv)
{
  printf("%d\n", gcd(120, 180)) ;
  return 0 ;
}
int gcd(int a, int b)
{
  return b?gcd(b,a%b):a ;
}
```

★ exercise-13-4 与えられた文字列を「逆順に」出力するプログラムを再帰を用いて書きなさい.

この問題は, 以前に例示した strrev を再帰を用いて実現するのではなく, 関数内部で出力を行うプログラムを書きなさいということです.

```
/* $Id: exercise13-4.c,v 1.1 2004-07-02 11:03:03+09 naito Exp $ */
#include <stdio.h>
#include <strings.h>
void *strrev(char *) ;
int main(int argc, char **argv)
{
  strrev("0123456789") ; printf("\n") ;
  return 0 ;
}
void *strrev(char *s)
{
  if (strlen(s) == 1) {
    printf("%c", *s) ;
    return ;
  }
  strrev(s+1) ;
  printf("%c", *s) ;
  return ;
}
```

★ exercise-13-5 （**難？**）次の仕様をみたす関数をつくりなさい.

【形式】

```
char *strrev(char *s)
```

【機能説明】

文字列 s を逆転した文字列に書き換えます. 戻り値は s へのポインタです.


```
/* $Id: exercise13-5.c,v 1.2 2004-07-02 11:07:46+09 naito Exp $ */
#include <stdio.h>
#include <strings.h>
char *strrev(char *) ;
int main(int argc, char **argv)
{
  char  s[]="0123456789" ;
  strrev(s) ; printf("%s\n",s) ;
  return 0 ;
}
char *strrev(char *s)
{
  char *e, c ;
  if (!*s) return NULL ;
  e = s+strlen(s)-1 ;
  if (e-s == 1) return s ;
  c = *e ; *e = '\000' ;
  strrev(s+1) ;
  *e = *s ; *s = c ;
  return s ;
}
```

【注意】 このプログラムは, 久保氏から「タコ」なプログラムだと言われました. 理由は, strrev を呼び出すたびに strlen を用いているため, $O(n^2)$ のプログラムになっているからです.

これを改良することは, 以下の方法をのいずれかを用いれば容易です.

1. 一旦 strrev を呼び出して, 文字列の長さも使った関数を再帰的に呼び出す.
2. 文字列の長さを strrev の内部に static 変数として保存しておく.

☆ exercise-13-6-3 再帰を用いてハノイの塔を解く手順を出力するプログラムを書きなさい.

```
/* $Id: exercise13-6.c,v 1.2 2004-07-02 11:10:04+09 naito Exp $ */
#include <stdio.h>
void    hanoi(int, int) ;
int main(int argc, char **argv)
{
  hanoi(10,0) ;
  return 0 ;
}
/* n: 板の枚数, p: それがある pole の番号
 * p = 0, 1, 2
 * (p+(n&1)+1)%3:
 *      if n:even: 0->1, 1->2, 2->0
 *      if n:odd : 0->2, 1->0, 2->1 */
void hanoi(int n, int p)
{
  if (n <= 0) return ;
  if (n == 1) {
    printf("%2d: %d -> %d\n", n, p, (p+(n&1)+1)%3) ;
    return ;
  }
  hanoi(n-1, p) ;
  printf("%2d: %d -> %d\n", n, p, (p+(n&1)+1)%3) ;
  hanoi(n-1, (p+((n-1)&1)+1)%3) ;
  return ;
}
```