3 再帰関数とデータ構造

簡単な再帰関数

```
# let rec length l =
    match l with
      [] -> 0
    | a::l -> 1 + length l ;;
val length : 'a list -> int = <fun>
# let length l =
    let rec count l n =
      match l with
        [] -> n
      | a::l -> count l (n+1)
      in
      count l 0 ;;
val length : 'a list -> int = <fun>
# let rec sum l =
    match l with
      [] -> 0
    | a::l -> a + sum l ;;
val sum : int list -> int = <fun>
# let rec nth l n =
    match l with
      [] -> failwith "nth"
    | a::l -> if n = 0 then a else nth l (n-1) ;;
val nth : 'a list -> int -> 'a = <fun>
# nth [] 2;;
Exception: Failure "nth".
```

再帰関数の正しさの証明は帰納法を使う.

定理 1 任意のリスト $\mathtt{l} = [a_1; \ldots; a_n]$ に対して，$\mathtt{sum\ l} \rightsquigarrow \sum_{k=1}^{n} a_k$.

証明 l の長さに関する帰納法を使う.

- $n = 0$ のとき，要素がないので，結果は 0 で正しい.
- 定理が n でなりたつとき，$n+1$ について証明する．$\mathtt{l}' = [b_1; \ldots; b_{n+1}] = \mathtt{a} :: \mathtt{l}$ とする．すなわち，$\mathtt{a} = b_1$ そして $\mathtt{l} = [b_2; \ldots; b_{n+1}]$．帰納法の仮定により，$\mathtt{sum\ l} \rightsquigarrow \sum_{k=2}^{n+1} b_k = s$．関数の定義により，$\mathtt{sum\ l}' \rightsquigarrow a + s = \sum_{k=1}^{n+1} b_k$.

蓄積引数があるともう少し複雑になる

定理 2 任意のリスト $\mathtt{l} = [a_1; \ldots; a_m]$ と整数 n に対して，$\mathtt{count\ l\ n} \rightsquigarrow m + \mathtt{n}$.

証明 l の長さに関する帰納法を使う.

- $m = 0$ のとき，要素がないので，結果は $0 + \mathtt{n} = \mathtt{n}$ で正しい．
- 定理が m でなりたつとき，$m+1$ について証明する．$\mathtt{l}' = [b_1; \ldots; b_{m+1}] = \mathtt{a} :: \mathtt{l}$ とする．すなわち，$\mathtt{a} = b_1$ そして $\mathtt{l} = [b_2; \ldots; b_{m+1}]$．帰納法の仮定により，count l $(\mathtt{n}+1) \rightsquigarrow m+(\mathtt{n}+1)$．関数の定義により，sum $\mathtt{l}'$ n $\rightsquigarrow m + (\mathtt{n}+1) = (m+1) + \mathtt{n}$．

リストと配列の変換

```
# let list_of_array arr =
    let len = Array.length arr in
    let rec list i =                          (* 結果が i 以降の要素から作ったリスト *)
      if i >= len then [] else arr.(i) :: list (i+1)
    in
    list 0 ;;
val list_of_array : 'a array -> 'a list = <fun>
# let list_of_array arr =
    let rec list l i =                        (* l が i+1 以降の要素から作ったリスト *)
      if i < 0 then l else list (arr.(i) :: l) (i-1)
    in
    list [] (Array.length arr - 1) ;;
val list_of_array : 'a array -> 'a list = <fun>
# let rec array_of_list l =                              (* 単純な帰納法を使う *)
    match l with
      [] -> [||]
    | a::l -> Array.append [|a|] (array_of_list l) ;;
val array_of_list : 'a list -> 'a array = <fun>
# let array_of_list l =                             (* Array.init で一気に作る *)
    Array.init (length l) (fun i -> nth l i) ;;
val array_of_list : 'a list -> 'a array = <fun>
# let rec list_to_array l arr i =                    (* list_of_array の逆処理 *)
    match l with
      [] -> arr
    | a::l -> arr.(i) <- a; list_to_array l arr (i+1) ;;
val list_to_array : 'a list -> 'a array -> int -> 'a array = <fun>
# let array_of_list l =
    let arr = Array.create (length l) 0 in               (* 分からないから 0? *)
    list_to_array l arr 0 ;;
val array_of_list : int list -> int array = <fun>                (* 型が違う! *)
# let array_of_list l =
    match l with
      [] -> [||]                                      (* 長さ 0 の配列を特別扱い *)
    | a :: l ->
        let arr = Array.create (length l + 1) a in
        list_to_array l arr 1 ;;            (* arr.(0) は既に正しいので，1 から *)
val array_of_list : 'a list -> 'a array = <fun>
```

データ構造の定義

```
# type family = Diamond | Heart | Clover | Spade ;;
# type value = King | Queen | Jack | Number of int ;;
# type card = Card of family * value | Joker ;;
# let ace = Card(Spade, Number 1);;
val ace : card = Card (Spade, Number 1)
# let stronger_value c1 c2 =
    match c1, c2 with
      _, King -> false                                  (* King に勝つものはない *)
    | King, _ -> true                                 (* King は King 以外の全てに勝つ *)
    | _, Queen -> false                           (* King 以外で Queen に勝つものはない *)
    | Queen, _ -> true                        (* Queen は King と Queen 以外の全てに勝つ *)
    | _, Jack -> false                                                  (* ... *)
    | Jack, Number _ -> true                          (* 相手はもう数字しかない *)
    | Number n1, Number n2 -> n1 > n2                 (* 数字ならその比較でいい *)
  ;;
val stronger_value : value -> value -> bool = <fun>
# let stronger c1 c2 =
    match c1, c2 with
      _, Joker -> false
    | Joker, Card _ -> true
    | Card(f1,v1), Card(f2,v2) -> stronger_value v1 v2
  ;;
val stronger : card -> card -> bool = <fun>
```

再帰的なデータ構造

```
# type 'a group =                                                  (* 群の上の式 *)
      Val of 'a
    | Neg of 'a group
    | Add of 'a group * 'a group ;;
# let e = Neg(Add(Val 3, Val 5)) ;;
val e : int group = Neg (Add (Val 3, Val 5))
# type 'a group_ops = {neg: 'a -> 'a; add: 'a -> 'a -> 'a};;    (* 群の作用子 *)
# let int_ops = {neg = (fun x -> -x); add = (fun x y -> x+y)};;
val int_ops : int group_ops = {neg = <fun>; add = <fun>}
# let rec eval ops e =
    match e with
      Val a -> a
    | Neg e -> ops.neg (eval ops e)
    | Add (e1, e2) -> ops.add (eval ops e1) (eval ops e2) ;;
val eval : 'a group_ops -> 'a group -> 'a = <fun>
# eval int_ops e ;;
- : int = -8
```

実習課題 (3 回目)

1. もっと簡単な stronger の定義として，各 card に整数の強さを与えればいい．例えば King は 13，Joker は 15．そうすると整数の比較で済む．その関数を定義せよ．
   `val strength : card -> int`

2. type 'a body (体における式)，type 'a body_ops，およびそのための eval を定義せよ．float で実験してみよ．

3. リストと配列のどちらでも行列の転置を定義せよ．
   `val list_transpose : 'a list list -> 'a list list`
   `val array_transpose : 'a array array -> 'a array array`
   それぞれの行列の表現は以下のとおり．
   [[a_{11}; ...; a_{1n}]; ...; [a_{m1}; ...; a_{mn}]]
   [| [|a_{11}; ...; a_{1n}|]; ...; [|a_{m1}; ...; a_{mn}|] |]
   配列の場合，Array.create_matrix m n a0 でも作れる．
   リストの場合では再帰関数を使え．長さ m のベクトルを m 行の行列に加える関数を定義すると便利．List.map と List.map2 も役に立つ．
   配列の場合では Array.init が便利．

4. リストと配列のどちらでも行列の積を定義せよ: $c_{ij} = \sum_k a_{ik} \cdot b_{kj}$
   `val list_prod : float list list -> float list list -> float list list`
   `val array_prod : float array array -> float array array -> float array array`
   要素の積をパラメーターに取った定義もできますか？

金曜日までに 感想や質問のできた分だけを computer-lecture-2005-aw-4@math.nagoya-u.ac.jp 宛に送って下さい．