1 整列 (続)

安定した整列 merge_sort

もう一つの基本的な考え方は，要素をリストに分けるのではなく，既に整列された二つのリストを融合することである．

```
# let rec merge l1 l2 =
    match l1, l2 with
      [], _ -> l2
    | _, [] -> l1
    | a :: l1', b :: l2' ->
        if a < b then a :: merge l1' l2 else b :: merge l1 l2'
  ;;
val merge : 'a list -> 'a list -> 'a list
# let rec merge_one ll =
    match ll with
      l1 :: l2 :: ll' -> merge l1 l2 :: merge_one ll'
    | _               -> ll ;;
val merge_one : 'a list list -> 'a list list
# let rec merge_all ll =
    match ll with
      []  -> []
    | [l] -> l
    | _   -> merge_all (merge_one ll) ;;
val merge_all : 'a list list -> 'a list
# let merge_sort l =
    merge_all (List.map (fun x -> [x]) l) ;;
val merge_sort : 'a list -> 'a list
merge_sort [3; 1; 5; 2] ;;
- : int list = [1; 2; 3; 5]
```

こちらの比較の回数を数えると，merge に関して最高で l1 と l2 の長さの合計である (最低でもそのどちらかの長さ)．merge_one では全ての部分リストに対してそれを行うので，元のリストの長さに比例する．merge_all が繰り返される回数は $\log_2 n$ なので，全体では

$$\text{比較の回数} = O(n \log_2 n)$$

挿入整列や選択整列よりはだいぶ早い．偶然に早くなることはないが，平均ではるかに勝っている．また，平均は quicksort と同じだが，最悪の場合でも安定しているのが長所である．

ちなみに，作られるリストのセルの数は比較の回数に比例するが，最初だけ各要素に対してセルが作られる．ということは，セルの数が $O(n(\log_2 n + 1))$．$\log_2 n$ 自体が小さな数になることが多いので，この差は無視できない．

練習問題 1 最初に List.map で一個だけのリストを作る代りに，merge_one みたい元のリストを 2 個ずつつみながら，2 個の整列したリストのリストを作る関数 merge_start を書きなさい．

```
val merge_start : 'a list -> 'a list list
```

2 再帰的データ構造

型略称の定義

```
# type nat = unit list ;;                                    (* 型の略称を定義する *)
type nat = unit list
# let succ (n : nat) : nat = () :: n;;
val succ : nat -> nat = <fun>
```

データ型の定義 略称とは別に，本当に新しい型も定義できる．

```
# type sort = Heart | Diamond | Spade | Club ;;    (* 構成子は大文字で始まる *)
# type figure = King | Queen | Jack | Number of int ;;
# type card = Joker | Card of sort * figure ;;           (* Card の引数は対 *)
# let joker = Joker ;;
val joker : card = Joker
# let heart_ace = Card (Heart, Number 1) ;;
val heart_ace : card = Card (Heart, Number 1)
# let strength card =
    match card with                                  (* パターンマッチングが使える *)
      Joker -> 15
    | Card (_, Number 1) -> 14
    | Card (_, King)     -> 13
    | Card (_, Queen)    -> 12
    | Card (_, Jack)     -> 11
    | Card (_, Number n) -> n
  ;;
val strength : card -> int
# strength heart_ace;;
- : int = 14
# strength (Card (Club, Number(-5)));;
- : int = -5
```

再帰データ型 型定義は再帰的であってもいい．

```
# type nat = Zero | Succ of nat ;;
# let rec add_nat n1 n2 =
    match n1 with
      Zero   -> n2
    | Succ n -> Succ (add_nat n n2)
  ;;
val add_nat : nat -> nat -> nat
# add_nat (Succ (Succ Zero)) (Succ Zero) ;;
- : nat = Succ (Succ (Succ Zero))
```

練習問題 2 上記の自然数の定義を使って，引き算と掛け算を定義しなさい．

リスト

リストも自分で定義できる．

```
# type 'a mylist = Nil | Cons of 'a * 'a mylist ;;          (* 'a は型引数 *)
# let nums = Cons (1, Cons (2, Nil)) ;;
val nums : int mylist = Cons (1, Cons (2, Nil))
# let rec length l =
    match l with
```

```
      Nil -> 0
    | Cons (_, l') -> 1 + length l' ;;
val length : 'a mylist -> int
# length nums ;;
- : int = 2
```

再帰データ型が将来のコードの変更を安全にしてくれる.

```
# type 'a mylist = Nil | Cons of 'a * 'a mylist | One of 'a (* mylist の定義を変更 *)
# let rec length l =                                    (* 元の定義をそのまま *)
    match l with
      Nil -> 0
    | Cons (_, l') -> 1 + length l'
  ;;
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
One _
val length : 'a mylist -> int = <fun>
```

間違ったプログラムがコンパイルできるものの，問題箇所を指摘したワーニングが出力される. (エラーにもできる)

木構造

リストだけではなく，様々な木構造が表現できる.

```
# type 'a tree = Empty | Node of 'a tree * 'a * 'a tree ;;
# let rec depth t =
    match t with
      Empty -> 0
    | Node (t1, x, t2) -> 1 + max (depth t1) (depth t2)
  ;;
val depth : 'a tree -> int
# depth (Node (Node (Empty, 1, Empty), 2, Empty)) ;;
- : int = 2
```

3 探索アルゴリズム

直線探索

```
let rec assoc (x : 'a) (l : ('a * 'b) list) =
  match l with
    [] -> raise Not_found
  | (a,b)::l ->
      if x = a then b else assoc x l
val assoc : 'a -> ('a * 'b) list -> 'b = <fun>
# assoc 2 [3,"c"; 1,"a"; 2,"b"];;
- : string = "b"
```

長所： 要素が簡単に追加できる $O(1)$
短所： 探索が遅い $O(n)$

例外 上の **raise** は例外を起こす関数である．例外は exn 型の値であり，様々な種類がある．Not_found はその一例である.

二分探索

```
let search (x : int) (arr : ('a * 'b) array) =
  let rec search first last =
    if first = last then
      let (a,b) = arr.(first) in
      if x = a then b else raise Not_found
    else
      let middle = (first + last) / 2 in
      let (a,_) = arr.(middle) in
      if x <= a then search first middle
      else search (middle+1) last
  in search 0 (Array.length arr + 1)
val search : int -> (int * 'a) array -> 'a = <fun>
# search 3 [|1,"a"; 2,"b"; 3,"c"; 4,"d"|] ;;
- : string = "c"
```

長所： 探索が早い $O(n \log n)$
短所： 追加の度に整列しなければならない $O(n)$

二分木探索

上に定義した木構造は効率のよい検索に向いている．そのために各節を見るだけで探しているものがどちらの子にあるかを知らなければなない．順序付けされた木では，左の子の要素が全て節の値より小さく，右の子の要素が全て節の値より大きい．

```
let rec search x t =
  match t with
    Empty -> raise Not_found
  | Node (left, (key, data), right) ->
      if x = key then data else
      if x < key then search x left else search x right
val search : 'a -> ('a*'b) btree -> 'b = <fun>

let rec add x d t =
  match t with
    Empty -> Node (Empty, x, d, Empty)
  | Node (left, (key, data as kd), right) ->
      if x < key then Node (add x d left, kd, right) else
      if x > key then Node (left, kd, add x d right) else
      Node (left, (key, d), right)
val add : 'a -> 'b -> ('a*'b) btree -> ('a*'b) btree = <fun>
```

長所： バランスが取れていれば，追加も検索も早い $O(\log n)$
短所： バランスがくずれると遅くなる 最悪 $O(n)$

検索の効率をよくするために，木の深さを最小に抑えないといけない．深さ n の木には 2^n 要素が入る．

練習問題 3 整列されたリストを元に，深さ $\log_2 n$ の整列された木を作る関数を書きなさい．先にリストを特定の長さで二つに切る関数を定義するとよい．