Jacques Garrigue, 2007 年 11 月 7 日

再帰関数プログラミング 3

8 リストの整列

集合の要素をある順序で並べるという操作は様々なところで必要になる．(可変な) 配列に対して行うこともできるが，不変なリスト構造に対しても同じぐらい効率よくできる．

挿入による整列

人間が自然に行う整列の一つである．元のリストから要素を順に取って行き，それを新しいリストの正しい所に挿入して行く．リストならではのやりかたでもある．

```
# let rec insert l x =
    match l with
      [] -> [x]
    | a::l' ->
        if x < a then x :: l else a :: insert l' x ;;
val insert : 'a list -> 'a -> 'a list
# let insertion_sort l =
    List.fold_left insert [] l ;;
val insertion_sort : 'a list -> 'a list
# insertion_sort [3; 1; 5; 2] ;;
- : int list = [1; 2; 3; 5]
```

とても分かりやすいやりかただが，調べて見ると効率があまりよくない．例えば，比較の回数を数えると，元々整列されていたリストに対して行うと，毎回構築して行くリストの全ての要素と比較しないと行けない．n がリストの長さなら，

$$\text{比較の回数} = \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = O(n)$$

作らなければいけないリストのセルの数も同様に $O(n^2)$ である．

とても運がよいときは入力が逆順になっていると，n 回で済むが，全ての順序に対する平均を考えるとやはり $O(n^2)$ になる．

練習問題 8.1 元のリストが整列されていたということはよくあるので，そのときは n 回で済むように定義を簡単に変えられるか？そうすると逆順の場合はどうなるか？

融合による整列 もう一つの基本的な考え方は，要素をリストに挿入するのではなく，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．

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 8.2 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
```

9 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 9.1 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
```

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らない．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 ordered lo t hi =
    match t with
      Empty -> true
    | Node (t1, a, t2) ->
        lo < a && a < hi && ordered lo t1 a && ordered a t2 hi ;;
val ordered : 'a -> 'a tree -> 'a -> bool
# let ordered_int t = ordered min_int t max_int ;;
val ordered_int : int tree -> bool
# let rec build_ordered l =
    match l with
      [] -> Empty
    | a :: l' ->
        let (l1, l2) = List.partition (fun x -> x < a) l' in
        Node (build_ordered l1, a, build_ordered l2) ;;
val build_ordered : 'a list -> 'a tree
# let t1 = build_ordered [3; 2; 1; 4] ;;
- : int tree =
Node (Node (Node (Empty, 1, Empty), 2, Empty), 3, Node (Empty, 4, Empty))
# ordered_int t1 ;;
- : bool = true
# let rec mem_tree x t =
    match t with
      Empty -> false
    | Node (t1, a, t2) ->
        a = x || if x < a then mem_tree x t1 else mem_tree x t2 ;;
val mem_tree : 'a -> 'a tree -> bool
```

ただし，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 9.2 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．