

ソフトウェア技法: No.4 (リストとリスト上の再帰関数)

亀山幸義 (筑波大学情報科学類)

1 リスト (list)

リストは、 n 個の (任意個の) 同種のデータの「並び」を表すデータ構造であり、関数型言語をはじめ多くのプログラム言語で用意されている。その特徴は、「要素を 1 つ追加する」、「先頭要素を取る」といった操作が定数時間^{*1}でできることである。

リストに似たデータ構造である「配列 (array)」や「組 (tuple)」では、「要素を 1 つ追加する」ためのコスト (計算時間) は小さくない。一方で、配列は、その「 n 番目の要素」を得るのが定数時間で済むが、リストでは、(その多くの実装では)、「 n 番目の要素」を得るのに n に比例する時間がかかってしまう。従って、これらのうちの操作を多く使うかによって、どちらのデータ構造を使うかを定めるのがよい。OCaml は配列も組も持っており、それらは後でまとめて紹介する。

リストで表すと良いデータは、処理の途中で、長さが伸び縮みするデータである。たとえば、数式 (n 次多項式など)、論理式の論理積標準形などを扱うときは、リストで表現するのが適している。一方で、行列やベクトルの計算であれば、(行列やベクトルのサイズが計算の途中で変わることは滅多にないので) 配列で表現するのが良いだろう。

この章では、リストに対する操作と、リストを操作する再帰的関数について述べる。

まず、OCaml でのリスト表現とリスト構成法を学ぼう。

```
(* 空のリスト *)
let list0 = [] ;;

(* 2つのブラケットをくっつける必要はない *)
let list0 = [ ] ;;

(* 5個の整数型の要素をもつリスト。要素の区切りは OCaml では、セミコロンである *)
let list1 = [1; 2; 3; 4; 5] ;;

(* 3個の文字列型の要素をもつリスト。 *)
let list2 = [ "this "; "is "; "a list." ] ;;

(* リストの先頭に要素を追加する (リストの cons 操作と呼ぶ) *)
let list3 = 8 :: list1 ;;

let list4 = "Wow! " :: list2 ;;

(* リストの要素は全て同じ型でなければいけないので、以下はエラー *)
let _ = "numbers" :: list1 ;;
```

ここまでのところで、リストの一般形は、 $a1 :: (a2 :: \dots (an :: []))$ あるいは、 $[a1; a2; \dots ;an]$ であることがわかったとおもいますが、実は、これらはまったく同一である。ここがリストのデータ構造で、ただひとつわかりにくいところであるが、要するに、本来は、 $a1 :: (a2 :: \dots (an :: []))$ というデータ構造なのだが、書くのがしんどいので、これを、 $[a1; a2; \dots ;an]$ と表す (省略形に過ぎない)。

*1 「定数」時間の処理とは、リストの大きさによらず一定の時間でできる処理という意味である。

```

(* リストの2つの表現が本当に同じか? *)
let _ = (1 :: (2 :: (3 :: [])) = [1; 2; 3]) ;;

(* :: の記号は右結合的である *)
let _ =
  if (1 :: (2 :: (3 :: [])) = 1 :: 2 :: 3 :: []) then
    "ok" else "ng" ;;

(* ところで、くどいようだが、:: の右に来るのは、リストの要素でなくリス
トそのものである。したがって :: の右に「要素のみ」を書くとエラー。 *)
let _ = 1 :: 2 ;; (* これはエラー *)

(* :: はあくまで、要素をリストにつなげる (非対称な) 操作である *)
let _ = 1 :: [2; 3] ;; (* これはOK*)

```

次に、リストを分解する操作を見てみよう。

```

(* リストの先頭要素を取るのは List モジュールの hd 関数である *)
(* この関数名は head に由来する *)
let _ = List.hd list1 ;;

let _ = List.hd list2 ;;

(* リストの先頭要素を除いた残りを取るのは List モジュールの tl 関数である *)
(* この関数名は tail に由来する *)
let _ = List.tl list1 ;;

let _ = List.tl list2 ;;

(* 空リストの head や tail を取るのはエラーである *)
let _ = List.hd list0 ;; (* エラー *)

let _ = List.tl list0 ;; (* エラー *)

```

ここでは hd, tl 関数を使ったが、リストを分解するためには、パターンマッチによる方法がある。

(* 空リストかどうかを判定する *)

```
let is_empty_list x =  
  match x with  
  | [] → true  
  | h::t → false ;;
```

(* 先頭要素に1を足した数を返す。ただし、空リストのときは0を返す *)

```
let add1_to_head x =  
  match x with  
  | [] → 0  
  | h::t → h + 1 ;;
```

(* 先頭要素を取り除いたリストを返す。ただし、空リストのときはエラーでなく空リストを返す *)

```
let tail2 x =  
  match x with  
  | [] → []  
  | h::t → t ;;
```

これらの例で、`match e1 with ...` はパターンマッチを行うための構文である。`e1` の部分にはパターンマッチをしたい式を書く。この式を計算した結果 (値) が、パターンマッチの対象物となる。`with` のあとの「...」の部分は、`p -> e` の形を、縦棒でいくつか並べた形をしている。(上記の例では2つだけだが、何個でもよい。) `p` はパターンとよばれ、式ではなく、変数、定数、それら`::`などで結合した形等をしている。`e1` を計算した結果 (値) と、パターン `p` とをマッチさせて、マッチに成功したら、`p -> e` の部分の式を評価して全体の計算結果とする。ただし、この `p` の部分とのマッチにおいて、変数が束縛されることがある (たとえば、上記で `[]` というパターンは変数を含まないが、`h::t` というパターンは変数 `h,t` を含むので、パターンマッチに成功したら、`h` と `t` は何らかの値をもつ。

パターンについての補足: OCaml では、パターンとして、割と複雑なものが書ける。たとえば `[x; y; z]` と書けば、長さ3のリストとのパターンマッチをあらわすので、長さ2や長さ5のリストとはマッチしない。ただし、パターンマッチするのは、あくまで (`match` と `with` の間にある式を計算した結果としての) 値であり、式ではないので、`x + y` のようなパターンはない。(このような式とパターンマッチすることはできない。) また、`[x; x]` のように「同じ要素が2個含まれるリスト」を書きたくなることもあるが、これも OCaml パターンとしては書けない。(このようなものを書きたいときは、`[x; y] when x=y` といった条件付きパターンを使うとよいが詳細は省略)。このようにパターンとパターンマッチの話はつきないので、詳細はマニュアル等を参照されたい。

パターンマッチは、上から順番に試される。つまり、上の方でマッチしたら、下の方でもマッチするとしても下の方は無視される。

また、OCaml 処理系は、パターンマッチにおいて、「すべての式がどれかのパターンとマッチするかどうか」をチェックしている。これを「網羅性チェック」と言う。網羅性チェックにひっかかった場合は、Warning(警告)を出す。Warning はエラーではないので無視してもよいが、自分の設計ミスに気付く良いきっかけになる。

(* 網羅性チェックにひっかかるので警告が出る *)

```
let foo x =  
  match x with  
  | h :: t → t ;;
```

(* 不足するケースを補えば警告が出なくなる *)

```
let foo x =  
  match x with  
  | h :: t → t  
  | [] → [] ;;
```

2 リスト上の再帰関数

さて、いよいよ本題であるリスト上の再帰関数の話である。

まずは、リストの長さを計算してみよう。length 関数は、List モジュールにはいっているので List.length という名前では呼べがよいが、ここでは自分で定義してみる。

(* パターンマッチを使わない書き方 *)

```
let rec length lst =  
  if lst = [] then 0  
  else length (List.tl lst) + 1 ;;
```

(* パターンマッチを使う書き方 *)

```
let rec length lst =  
  match lst with  
  | [] → 0  
  | h :: t → length t + 1 ;;
```

length 関数は明らかにループ構造が必要であるので、再帰関数として実装した。ここでの再帰関数の作り方は、前に述べた自然数に対する再帰関数と同じ方針であり、「漸化式」を思い浮かべるものである。すなわち、「リスト lst の長さ」は、「(List.tl lst) の長さ」に 1 を足したものである。

このように、リストにおいては、漸化式で必要となる「1 つ小さいリスト」として「List.tl lst」を考えると良いことが多い。

ところで、細かいことであるが、上記の 2 つ目の length の定義におけるパターンマッチでは、パターンに含まれる変数 h を右辺で使っていない。そこで、前にでてきた underscore に置きかえた方がよい。

(* パターンにおいて underscore を使う書き方 *)

```
let rec length lst =  
  match lst with  
  | [] → 0  
  | _ :: t → length t + 1 ;;
```

前のプログラムと意味的には同じであるが、この方が無駄な変数を使っておらず、OCamlらしいプログラムである。

次は、整数のリストに対する再帰関数である。

(* 整数リストに対して、その総和を求める *)

```
let rec sumlist lst =  
  match lst with  
  | []      → 0  
  | h :: t  → h + sumlist t ;;
```

(* 整数リストに対して、 n より大きい要素だけを残す *)

```
let rec filter_greater_than n lst =  
  match lst with  
  | []      → []  
  | h :: t  → if h > n then h :: filter_greater_than n t  
              else filter_greater_than n t ;;
```

(* 整数リストに対して、それが昇順(小さいものが前)になっているかどうか判定する *)

(* いきなり書けないので、補助関数を定義する *)

```
let rec sorted_sub x lst =  
  match lst with  
  | []      → true  
  | h :: t  → (x <= h) && (sorted_sub h t) ;;
```

(* 昇順チェック自体は再帰でないので `rec` はつけない *)

```
let sorted lst =  
  match lst with  
  | []      → true  
  | h :: t  → sorted_sub h t ;;
```

(* 昇順になっている整数リストの適切な場所に x の値を挿入したリストを作る *)

```
let rec insert x lst =  
  match lst with  
  | []      → [x]  
  | h :: t  → if x <= h then (* ここにいれればよい! *)  
              x :: lst  
              else h :: (insert x t)
```

ここまで来ると、リストの操作は自由自在にできるであろう。

(* 整数リストに対して、その要素をそれぞれ二乗したリストを求める *)

```
let rec map_square lst =  
  match lst with  
  | []      → []  
  | h :: t  → (h * h) :: (map_square t) ;;
```

(* 整数リストに対して、その要素の二乗の総和を求める *)

```
let rec square_sum lst =  
  match lst with  
  | []      → 0  
  | h :: t  → (h * h) + (square_sum t) ;;
```

2つのリストをつなげる関数を `append` と言う。これも `List.append` として提供されているが、自分で定義してみよう。この場合、引数を `lst1` と `lst2` とすると、`lst1` に関する漸化式 (再帰) で解くのがよく、`lst2` に関する漸化式では解けないことに気付くと、あとはすらすらできる。

```
(* 2つのリストの結合 (append) *)
let rec append lst1 lst2 =
  match lst1 with
  | []      → lst2
  | h :: t → h :: (append t lst2) ;;
```

さらに、リストの順番を逆転したリストを作る関数 `reverse` を定義しよう。

```
(* これでは型があわない。エラー *)
let rec reverse_buggy lst =
  match lst with
  | []      → []
  | h :: t → (reverse_buggy t) :: h ;; (* 要素 :: リスト になっていない *)

(* そこで、append を利用することにする *)
let rec reverse lst =
  match lst with
  | []      → []
  | h :: t → append (reverse t) [h] ;;
```

上記で、一応 `reverse` はできたのであるが、実は効率が悪いものである。なぜなら、最後の行で `reverse t` を作ったあとに `append` しているが、`append` は第一引数のリストを分解しながら、1つずつ第2引数にくっつけていくからである。つまり、`append` の計算には、第一引数の長さに比例する時間がかかる。そうすると、上記の `reverse` の計算にかかる時間は、(最初に与えられたリスト `lst` の長さを n とすると)、 $(n-1) + (n-2) + \dots + 1$ に比例する時間がかかり、これは、 n の2次式、つまり、計算量は $O(n^2)$ となってしまうのである。さらに、メモリもかなり無駄に使っている。

これを改善するためには、`accumulating parameter` (蓄積引数、累積引数) と呼ばれる技法を使うとよい。

```
(* 蓄積引数 acc を使った reverse 計算の補助関数 *)
let rec reverse_sub lst acc =
  match lst with
  | []      → acc
  | h :: t → reverse_sub t (h :: acc) ;;

(* 効率良い reverse の本体 *)
let reverse_better lst =
  reverse_sub lst [] ;;
```

このプログラムの動作を1度は、手計算で追ってみてほしい。1つ目の関数 `reverse_sub` の計算で、引数 `acc` に「計算の途中結果」がどんどん蓄積されていくことがわかる。これが `accumulating parameter` の意味である。

`reverse_better` の計算は、1回の再帰呼び出しにおいて、(`append` でなく) `::` 演算しかやっていない (これはリストの長さによらずに定数時間である)。よって、`reverse_better` の計算は、与えられたリストの長さ n に対して、 $O(n)$ の時間で済んでおり、計算時間を大幅に改善できた。

3 寄り道

ここまでの範囲では、ユーザプログラムからエラーを起こす方法をやっていなかったのが、処理がいまひとつのところがあつた。たとえば、「先頭要素に1を足した数を返す。ただし、空リストのときは0を返す。」というのは、本来であれば、「先頭要素に1を足した数を返す。ただし、空リストのときはエラーにする。」というのが良い。同様に、「先頭要素を取り除いたリストを返す。ただし、空リストのときはエラーでなく空リストを返す。」というのも、「先頭要素を取り除いたリストを返す。ただし、空リストのときはエラー。」がよい。このために、OCaml では `failwith` という関数を使うことができる。

```
(* 先頭要素に1を足した数を返す。ただし、空リストのときはエラー *)
let add1_to_head2 x =
  match x with
  | [] → failwith "cannot take the first element of an empty list"
  | h::t → h + 1 ;;

(* 先頭要素を取り除いたリストを返す。ただし、空リストのときはエラー *)
let tail2 x =
  match x with
  | [] → failwith "cannot take the tail of an empty list"
  | h::t → t ;;
```

このように `failwith` は文字列型の引数を取る関数であり、それが呼ばれると、実行を (突然に) 止めて、エラーメッセージを出すものである。

4 演習問題

以下の問題から3つ以上を解きなさい。

- 与えられた整数リストの最大値を計算する関数 `max_list` を定義せよ。
例: `max_list [1; 5; 0; 4; 1; 0] = 5`
- 与えられた整数リストにおいて、0以外の要素の個数を数えよ。(そのような要素が重複して2回以上出現しても、その回数だけ数える。)
例: `num_nonzero [1; 5; 0; 4; 1; 0] = 4`
- 相異なる整数のリストが与えられたとき、その中で3番目に大きな値を求めよ。(長さが3未満のリストの場合は、`failwith` で異常終了せよ。)
[2017/7/12 誤植修正; `second` になっていたので `third` に直しました。]
例: `third_largest [1;10;3;6;2;7] = 6`
- 与えられたリストの最後の要素を返す関数を定義せよ。ただし、空リストに対しては、`failwith` 関数で異常終了せよ。
例: `last [1;10;3;6;2;7] = 7`, `last [] = (異常終了)`
- 整数のリストのリストが与えられたとき、それぞれのリストの総和のリストを返す関数を定義せよ。
例: `sum_list [[1;2;3]; [4;5]; [6;7;8;9;10]] = [6; 9; 40]`
- (発展課題) 昇順に並んでいる2つの整数リストが与えられたとき、それらを昇順に併合 (マージ) する関数 `merge` を作りなさい。
例: `merge [1; 3; 5] [2; 3; 10; 15] = [1; 2; 3; 3; 5; 10; 15]`
- (発展課題) 与えられたリストの、上位N番目の要素を返す関数 `nth_top` を作りなさい。
例: `nth_top 3 [1; 7; 5; 3; 4] = 4`, `nth_top 10 [1; 2; 3] = (異常終了)`