情報システム科学実習II第5回
再帰的多相的データ構造: リスト

担当: 山口 和紀・五十嵐 淳

2001年11月14日

1 リスト

これまで，構造のあるデータを表現するための手段としては組を用いてきた．ここでは**リスト**(*lists*) という,「データの (有限) 列」を表現するデータ構造をみていく．リストは構造自体が再帰的に定義され，また格納するデータの種類に制限がないという意味で多相的であるという特徴をもっている．

1.1 リストの構成法

まずは簡単なリストの例を見てみよう．リストはデータ列を構成する要素を ; で区切り，[] で囲むことで構成される．

```
# [3; 9; 0; -10];;
- : int list = [3; 9; 0; -10]
# let week = ["Sun"; "Mon"; "Tue"; "Wed"; "Thu"; "Fri"; "Sat"];;
val week : string list = ["Sun"; "Mon"; "Tue"; "Wed"; "Thu"; "Fri"; "Sat"]
```

リストの式には “⟨要素の型⟩ list” という型が与えられる．これが意味するのは「(OCaml では) 一つのリストに並ぶ要素の型は同じ」でなければならない，ということである．また別の見方をすると，リスト式はその列の長さに関わらず，要素の型が同じであれば，同じリスト型に属するということである．

```
# [1; 'a'];;
Characters 5-8:
This expression has type char but is here used with type int
# (* compare with the type of [3; 9; 0; -10] *)
# [-3];;
- : int list = [-3]
```

このことは組とリストの決定的な違いであるので注意すること．組の型は各要素の型を並べることで記述されるため，各要素の型は異ってもよいが，大きさの違う組は同じ型に属し得ない (すなわち，組の大きさの情報が型に現れている).

リストの構造は，組のように「要素を並べたもの」と思う代りに，以下のように再帰的に定義されている構造とも考えられる．つまり

- `[]` は，**空リスト**(*empty list, null list*) と呼ばれ，リストである．
- リスト l の先頭に (l の要素と同じ類いの) 要素 e を追加したもの (e `::` l と書く) もリストである．

と定義することもできる．この定義は二番目の節が再帰的になっている．`::` のことを **cons オペレータ**(または単に cons) と呼ぶこともある．このようにリストを構築する例を見てみよう．

```
# let oddnums = [3; 9; 253];;
val oddnums : int list = [3; 9; 253]
# let more_oddnums = 99 :: oddnums;;
val more_oddnums : int list = [99; 3; 9; 253]
# (* :: is right associative, that is, e1 :: e2 :: l = e1 :: (e2 :: l) *)
# let evennums = 4 :: 10 :: [256; 12];;
val evennums : int list = [4; 10; 256; 12]
# [];;
- : 'a list = []
```

要素を列記する方法と `::` を用いる方法を混ぜることもできる．`evennums` の例に見るように，`::` は右結合する (`e1 :: e2 :: l` は `e1 :: (e2 :: l)` と読む)． 空リスト `[]` は要素がなく，どのような要素のリストとも見なせること ができるため，`'a list` という多相型が与えられている．もちろん，空リストには様々な型の要素を追加することができる．

```
# let campuslist = "Komaba" :: "Hongo" :: [];;
val campuslist : string list = ["Komaba"; "Hongo"]
# let boollist = true :: false :: [];;
val boollist : bool list = [true; false]
```

ちなみに OCaml では「要素を並べる」定義は，(内部的には) 再帰的な定義の略記法である．つまり，

$$[e_1;\ e_2;\ \cdots\ ;\ e_n] = e_1 :: e_2 :: \cdots :: e_n :: []$$

である．

cons オペレータ `::` は，あくまで「一要素の (先頭への) 追加」を行うもので，リストにリストを追加 (連結) するという操作や，リストの最後尾へ要素を追加するといった操作は `::` で行えない．

```
# [1; 2] :: [3; 4];;
Characters 12-13:
This expression has type int but is here used with type int list
# [1; 2; 3] :: 4;;
Characters 13-14:
This expression has type int but is here used with type int list list
```

:: は文法キーワードなので，:: の型を見ることはできないが，敢えて型を考えるなら 'a -> 'a list -> 'a list と思うのがよいだろう．リストはどんな値でも要素にでき，関数のリスト，リストのリスト等を考えることも可能である．

```
# [(fun x -> x + 1); (fun x -> x * 2)];;
- : (int -> int) list = [<fun>; <fun>]
# [1; 2; 3] :: [[4; 5]; []; [6; 7; 8]];;
- : int list list = [[1; 2; 3]; [4; 5]; []; [6; 7; 8]]
```

2 番目の例は，整数リストのリストに整数リストを :: で追加している．

1.2 リストの要素へのアクセス: match 式とリストパターン

さて，リストの要素にアクセスするときには組と同様にパターンを用いる．リストパターンは

[⟨パターン$_1$⟩; ⟨パターン$_2$⟩; ...; ⟨パターン$_n$⟩]

で n 要素のリスト (n が 0 なら空リスト) にマッチさせることができる．また，:: を使ってパターンを記述することもできる．

⟨パターン$_1$⟩ :: ⟨パターン$_2$⟩

と書いて，先頭の要素を ⟨パターン$_1$⟩ に，**残りのリストを** ⟨パターン$_2$⟩ にマッチさせることができる．次の関数は，三要素の整数リストの要素の和を計算する関数である．

```
# (* equivalent to
#    let sum_list3 (x :: y :: z :: []) = x + y + z *)
# let sum_list3 ([x; y; z]) = x + y + z;;
Characters 85-108:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val sum_list3 : int list -> int = <fun>
```

リスト式の場合と同様，[x; y; z] というパターンは :: と [] を使ったパターンで表すことができる．ここで，重要なことはこの関数をコンパイルをするとコンパイラから警告が発せられていることである．この “nonexaustive pattern” と呼ばれる警告は,「パターンの表現に属する値でありながら，パターンにマッチしない値が存在する」という意味であり，コンパイラはマッチしない値の例を示してくれる．たしかにこの例では引数の型は int list であるにも関わらず三要素のリストにしかマッチしない．実際，マッチしない値に関数を適用するとマッチングに失敗したという例外が発生する．

```
# sum_list3 [4; 5; 6];;
- : int = 15
# sum_list3 [2; 4];;
Uncaught exception: Match_failure ("", 86, 109).
```

では，任意の長さの整数リストの要素の和を取る関数を書くにはどうすればよいのだろうか? 例え，二要素のリストの和を取る関数，四要素リストの和を取る関数などなどを定義し，それを何らかの手段で組み合わせることができたとしても，それでは不十分である．関数定義の大きさが無限に長くなってしまう! ここで，リストが再帰的な定義をされた構造であることが非常に重要な意味を持ってくる．つまり，リストの再帰的な定義から，要素の和を取る定義を導くことができるのである．それが以下の定義である．

- 空リストの全要素の和は 0 である．
- 先頭要素が n で，残りのリスト l に対する全要素の和を s とすると，n :: l の全要素の和は $n + s$ である．

ポイントは，長いリストの全要素の和はより短いリストの全要素の和から計算できることである．(fact などの定義と比べてみよ．) これを OCaml の定義とするためには，与えられたリストが空かそうでないかを判断する手段が必要であるが，ひとまず例を見て，その判断をどのように行うか見てみよう．

```
# let rec sum_list l =
#   match l with
#     [] -> 0
#   | n :: rest -> n + (sum_list rest);;
val sum_list : int list -> int = <fun>
```

まず，sum_list は再帰関数になっている． match 以下が l が空リストかそうでないかを判断し，別の処理を行って いる部分で match 式と呼ばれる．match 式の一般的な文法は，

match ⟨式$_0$⟩ with ⟨パターン$_1$⟩ -> ⟨式$_1$⟩ | ... | ⟨パターン$_n$⟩ -> ⟨式$_n$⟩

という形で与えられ，⟨式$_0$⟩ を評価した結果を，⟨パターン$_1$⟩ から**順番に**マッチさせていき，⟨パターン$_i$⟩ でマッチが成功したら，⟨式$_i$⟩ を評価する．全体の値は ⟨式$_i$⟩ の値である．各パターンで導入された変数 (n, rest) は対応する式の中だけで有効である．上の OCaml による定義が，自然言語による定義に対応していることを確かめよ．多くのリストを操作する関数は sum_list のように，空リストの場合の処理と，cons の場合の処理を match で組み合わせて書かれる．

match 式についての注意　match 式がマッチングを順番に行う，というのは非常に重要な点である．もしも，同じ値が複数のパターンにマッチする場合は先に書いてあるパターンにマッチしてしまう．このような例は特に**定数パターン**を使用すると発生しやすい．定数パターンは整数，文字列定数をパターンとして用いるもので，その定数にのみマッチする．また，前に書いてあるパターンのせいで，パターンにマッチする値がない場合，コンパイラは警告を発する．

```
# let f x =
#   match x with (1, _) -> 2 | (_, 1) -> 3 | (1, 1) -> 0 | _ -> 1;;
```

```
Characters 56-60:
Warning: this match case is unused.
val f : int * int -> int = <fun>
# f (1, 1);;
- : int = 2
```

(1, 1) は最初のパターンにマッチする．また，3 番目のパターンは決して使われないので警告が出ている．

さて，一般的なリスト操作関数の例を見ていく前に，もうひとつ例をみていこう．(空でない) 整数リストのなかから最大値を返す関数 max_list は

```
# let rec max_list l =
#   match l with
#     [x] -> x
#   | n1 :: n2 :: rest ->
#      if n1 > n2 then max_list (n1 :: rest) else max_list (n2 :: rest);;
Characters 24-144:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val max_list : 'a list -> 'a = <fun>
```

のように定義される．

1.3 リスト操作の関数

さて，リストに対する基本的な操作 (構成と要素アクセス) をみたところで，様々なリスト操作を行う関数を見ていこう．多くの関数がリストの構造に関して再帰的に定義される．また，ほとんどすべての関数が要素型によらない定義をしているため，多相型が与えられることに注意しよう．

hd, tl, null リストの先頭要素，リストの先頭を除いた残りを返す関数 hd (head の略), tl (tail の略) は以下のように定義され，

```
# let hd (x::rest) = x
# let tl (x::rest) = rest;;
Characters 8-21:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
Characters 29-45:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val hd : 'a list -> 'a = <fun>
val tl : 'a list -> 'a list = <fun>
```

空リストに対しては働けない nonexaustive な関数である．null は与えられたリストが空かどうかを判定する関数である．

```
# let null = function [] -> true | _ -> false;;
val null : 'a list -> bool = <fun>
```

function キーワードは fun と match を組み合わせて匿名関数を定義するもので，

function ⟨パターン$_1$⟩ -> ⟨式$_1$⟩ | ... | ⟨パターン$_n$⟩ -> ⟨式$_n$⟩

で

fun x -> match x with ⟨パターン$_1$⟩ -> ⟨式$_1$⟩ | ... | ⟨パターン$_n$⟩ -> ⟨式$_n$⟩

を示す．最後に受け取った引数に関して即座にパターンマッチを行うような関数定義の際に便利である．

null, hd, tl (と if) があれば match を使わずともリストを扱うことができる．

nth, take, drop　次は，n 番目の要素を取り出す nth, n 番目までの要素の部分リストを取り出す take，n 番目までの要素を抜かした部分リストを取り出す drop である．リストの要素は先頭を一番目とする．

```
# let rec nth n l =
#   if n = 1 then hd l else nth (n - 1) (tl l)
# let rec take n l =
#   if n = 0 then [] else (hd l) :: (take (n - 1) (tl l))
# let rec drop n l =
#   if n = 0 then l else drop (n - 1) (tl l);;
val nth : int -> 'a list -> 'a = <fun>
val take : int -> 'a list -> 'a list = <fun>
val drop : int -> 'a list -> 'a list = <fun>
```

これらの関数はリストの構造でなく，n に関しての再帰関数になっているので，上で見たパターンにあっていない．

```
# let ten_to_one = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1; 0];;
val ten_to_one : int list = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1; 0]
# nth 4 ten_to_one;;
- : int = 7
# take 8 ten_to_one;;
- : int list = [10; 9; 8; 7; 6; 5; 4; 3]
# drop 7 ten_to_one;;
- : int list = [3; 2; 1; 0]
# take 19 ten_to_one;;
Uncaught exception: Match_failure ("", 30, 46).
```

length　次はリストの長さを返す関数 length である．

```
# let rec length = function
#     [] -> 0
#   | _ :: rest -> 1 + length rest;;
val length : 'a list -> int = <fun>
```

length の型は _ パターンを使って先頭要素を使用してないことからわかるように，どんなリストに対しても使うことができる．実際，型をみると入力の要素型が型変数になっている．

```
# length [1; 2; 3];;
- : int = 3
# length [[true; false]; [false; false; false;]];;
- : int = 2
```

また length はふたつめの結果にみられるように，一番外側のリストの長さを計算するものである (結果は 5 ではない)．

append, reverse　次に示す append 関数はリスト同士を連結する関数である．append l1 l2 の再帰的定義は

- 空リストに l_2 を連結したものは l_2 である．
- 先頭が v で，残りが l'_1 であるリストに l を連結したものは，l'_1 と l の連結の先頭に v を追加したものである．

と考えることができる．

```
# let rec append l1 l2 =
#   match l1 with
#     [] -> l2
#   | x :: rest -> x :: (append rest l2);;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [1; 2; 3] [4; 5; 6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

この append の定義は，l1 の長さが長くなるほど再帰呼び出しが深く行われ，l2 の長さには関係がない．ちなみに append 関数は，もともと OCaml 起動時に中置オペレータ @ として利用可能である．

```
# [1; 2; 3] @ [4; 5; 6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

また append を使ってリストを反転させる reverse 関数を定義できる．

```
# let rec reverse = function
#     [] -> []
#   | x :: rest -> append (reverse rest) [x];;
val reverse : 'a list -> 'a list = <fun>
```

リストの最後に要素を追加することは直接はできないので，一要素リストを作って append している．しかし，この関数はあまり効率的ではない．なぜなら，reverse の呼び出し一度につき，append が一度呼ばれるが，この時 append の第一引数の長さは反転させようとする引数の長さ -1 であり append を計算するのにその長さ分の計算量を必要とする．reverse の再帰呼び出し回数は与えたリストの長さなので，リストの長さの自乗に比例した計算時間がかかってしまう．

これを改善したのが次の定義である．

```
# let rec revAppend l1 l2 =
#   match l1 with
#     [] -> l2
#   | x :: rest -> revAppend rest (x :: l2)
# let rev x = revAppend x [];;
val revAppend : 'a list -> 'a list -> 'a list = <fun>
val rev : 'a list -> 'a list = <fun>
```

最初の再帰関数 revAppend が第一引数を先頭から順に l2 に追加して行く関数である．先頭から追加していくため，l1 の順が逆になって l2 に連結される．

```
# revAppend [1; 2; 3] [4; 5; 6];;
- : int list = [3; 2; 1; 4; 5; 6]
```

この関数も append と同じく，第一引数の長さだけに比例した時間がかかる．リストの反転は revAppend の第二引数が空である特別な場合である．

```
# rev ['a'; 'b'; 'c'; 'd'];;
- : char list = ['d'; 'c'; 'b'; 'a']
```

map　map はリストの各要素に対して同じ関数を適用した結果のリストを求めるための高階関数である．

```
# let rec map f = function
#     [] -> []
#   | x :: rest -> f x :: map f rest;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

たとえば，整数リストの各要素を 2 倍する式は map を使って，

```
# map (fun x -> x * 2) [4; 91; 0; -34];;
- : int list = [8; 182; 0; -68]
```

と書くことができる．map の型は今まで見た中でかなり複雑である．まず，'a -> 'b で「何らかの関数」が第一引数であることがわかる．カリー化関数とみるならば，第二引数は「何らかの関数」の定義域の値を要素とするリストで，結果が「何らかの関数」の値域の値を要素とするリストとなる．または「何らかの関数」を与えた時点でリストからリストへの関数が返ってきていると解釈してもよい．

forall, exists　forall はリストの要素に関する述語 (要素から bool への関数) と，リストをとり，全要素が述語を満たすかどうかを，exists は同様に述語とリストをとって，述語を満たす要素があるかどうかを返す関数である．

```
# let rec forall p = function
#     [] -> true
#   | x :: rest -> if p x then forall p rest else false
# let rec exists p = function
#     [] -> false
#   | x :: rest -> (p x) or (exists p rest);;
val forall : ('a -> bool) -> 'a list -> bool = <fun>
val exists : ('a -> bool) -> 'a list -> bool = <fun>
# forall (fun c -> 'z' > c) ['A'; ' '; '+'];;
- : bool = true
# exists (fun x -> (x mod 7) = 0) [23; -98; 19; 53];;
- : bool = true
```

畳み込み関数 fold　上で見た sum_list, append はリストの要素すべてを用いた演算をするものである．実はこの二つの関数は共通の計算の構造を持っている．sum_list は [i1; i2; ...; in], つまり i1 :: i2 :: ... :: in :: [] から i1 + (i2 + (... + (in + 0)...)) を計算し，append [e1; e2; ...; en] l2 は e1 :: (e2 :: ... :: (en :: l2)...) を計算している．このふたつの計算の共通点は，

- 引数リストの cons を，sum_list では + で，append では :: 自身で置換え，
- 末尾の空リストも，sum_list では 0 で，append では l2 で置換え，
- 右から演算を順次行っていく

ことである．このような「右から畳み込む」計算構造を一般化した高階関数を fold_right と呼ぶ．逆に左から畳み込むのを fold_left と呼ぶ．rev は fold_left の例である．何故なら，rev [e1; e2; ...; en] は l ::: x を x :: l として，

```
(...(([] ::: e1) ::: e2) ... ::: en)
```

と表現できるからである．

以下が fold_right, fold_left の定義である．

```
fold_right f [e1; e2; ...; en] e ⟹ f e1 (f e2 (... (f en e)...))

fold_left f e [e1; e2; ...; en] ⟹ f (... (f (f e e1) e2) ...) en
```

を計算する．

```
# let rec fold_right f l e =
#   match l with
#     [] -> e
```

```
#   | x :: rest -> f x (fold_right f rest e)
# let rec fold_left f e l =
#   match l with
#     [] -> e
#   | x :: rest -> fold_left f (f e x) rest;;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# fold_right (fun x y -> x + y) [3; 5; 7] 0;;
- : int = 15
# fold_left (fun x y -> y :: x) [] [1; 2; 3];;
- : int list = [3; 2; 1]
```

fold_left, fold_right は要素はそのままで cons を適当な演算子に読み替えて，計算をするものと思うことができる．一方，map 関数はリストの構造はそのままで，要素だけを操作するような計算構造を抽象化した高階関数であった．実はリストに関する再帰関数はほとんど map と fold_left または fold_right を組み合わせて定義することができる．例えば，length は全要素をまず map を使って 1 に置換えて，足し算による畳み込みを行えばよいので，

```
# let length l = fold_right (fun x y -> x + y) (map (fun x -> 1) l) 0;;
val length : 'a list -> int = <fun>
```

と定義することも可能である．

1.4 Case Study: ソートアルゴリズム

リストを使ったソートアルゴリズムをいくつかみていこう．以下ではリストを < に関する昇順 (小さい方から順) に並べ替えることを考える．(比較演算子 < は多相的であるため，ソート関数も多相的に使える．)

まずは準備として，ソートの対象として用いる，疑似乱数列を生成するための関数を定義する．

```
# let nextrand seed =
#   let a = 16807.0 and m = 2147483647.0 in
#   let t = a *. seed
#   in t -. m *. floor (t /. m)
# let rec randlist n seed tail =
#   if n = 0 then (seed, tail)
#   else randlist (n - 1) (nextrand seed) (seed::tail);;
val nextrand : float -> float = <fun>
val randlist : int -> float -> float list -> float * float list = <fun>
# randlist 10 1.0 [];;
- : float * float list =
2007237709,
[1458777923; 1457850878; 101027544; 470211272; 1144108930; 984943658;
 1622650073; 282475249; 16807; 1]
```

挿入ソート(*insertion sort*) は，既にソート済のリストに新しい要素をひとつ付け加える操作を基本として，各要素を順に付け加えていくものである．基本操作 insert は

```
# let rec insert (x : float) = function
#     (* the second argument is already sorted *)
#     [] -> [x]
#   | (y :: rest) as l -> if x < y then x :: l else y :: (insert x rest);;
val insert : float -> float list -> float list = <fun>
# insert 4.5 [2.2; 9.1];;
- : float list = [2.2; 4.5; 9.1]
```

と書くことができる．パターン中に出現する

〈パターン〉 as 〈変数〉

は as **パターン**と呼ばれるもので，パターンにマッチした値全体を〈変数〉で参照できるものである．ここでは x :: y :: rest と書く代りに x :: l としている．この insert を使ってソートを行う関数は

```
# let rec insertion_sort = function
#     [] -> []
#   | x :: rest -> insert x (insertion_sort rest);;
val insertion_sort : float list -> float list = <fun>
```

と定義できる．

挿入ソートは insert, insertion_sort ともに入力に比例する回数の再帰呼出しを行うため，計算には与えられたリストの長さの自乗に比例した時間がかかる．**クイックソート**(*quick sort*) は C.A.R. ホーアが発明した効率の良いソートアルゴリズムで，**分割統治法**(*divide and conquer*) に基づき，下のような要領でソートを行う．

- 要素から適当にピボットと呼ばれる値を選びだす．
- 残りの要素をピボットに対する大小でふたつの集合に分割する．
- それぞれの部分集合にたいしてソートを行い，その結果を結合する．

```
# let rec quick = function
#     [] -> []
#   | [x] -> [x]
#   | x :: xs ->  (* x is the pivot *)
#       let rec partition left right = function
#           [] -> (quick left) @ (x :: quick right)
#         | y :: ys -> if x < y then partition left (y :: right) ys
#                      else partition (y :: left) right ys
#       in partition [] [] xs;;
val quick : 'a list -> 'a list = <fun>
```

この quick の定義は append を使用しているのでまだ効率の悪い点が残っている．(append を使用しない定義は練習問題．) クイックソートはリストの長さを n として，平均で $n \log n$ に比例した時間で行えることが知られている．insert_sort (snd (randlist 10000 1.0 [])) と quick (snd (randlist 10000 1.0 [])) を試してみよ．(snd はペアから第二要素を取り出す定義済の関数である．)

1.5 練習問題

Exercise 5.1 次のうち，正しいリスト表現はどれか．コンパイラに入力する前に，正しい場合と思うは型を，間違っていると思う場合はなぜ誤りか，を予想してから実際に確認せよ．

1. `[[]]`
2. `[[1; 3]; ["hoge"]]`
3. `[3] :: []`
4. `2 :: [3] :: []`
5. `[] :: []`
6. `[(fun x -> x); (fun b -> not b)]`

Exercise 5.2 sum_list，max_list を，match を使わず null, hd, tl の組合わせのみで定義せよ．match を使うテキストの定義と比べ，記述面などの利害得失を議論せよ．

Exercise 5.3 l2 が空リストの時に効率がよくなるよう append の定義を書き換えよ．

Exercise 5.4 次の関数を定義せよ．

1. 正の整数 n から 0 までの整数の降順リストを生成する関数 downto0.
2. 与えられた正の整数のローマ数字表現 (文字列) を求める関数 roman．(I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000 である．) ただし，roman はローマ数字の定義も引数として受け取ることにする．ローマ数字定義は，単位となる数とローマ数字表現の組を大きいものから並べたリストで表現する．例えば

```
roman [(1000, "M"), (500, "D"), (100, "C"), (50, "L"),
       (10, "X"), (5, "V"), (1, "I")] 1984
⟹ "MDCCCCLXXXIIII"
```

   4, 9, 40, 90, 400, 900 などの表現にも注意して，

```
roman [(1000, "M"), (900, "CM"), (500, "D"), (400, "CD"),
       (100, "C"), (90, "XC"), (50, "L"), (40, "XL"),
       (10, "X"), (9, "IX"), (5, "V"), (4, "IV"), (1, "I")] 1984
⟹ "MCMLXXXIV"
```

となるようにせよ.

3. 与えられたリストのリストに対し，内側のリストの要素を並べたリストを返す関数 concat.

```
concat [[0; 3; 4]; [2]; [5; 0]; []] = [0; 3; 4; 2; 5; 0]
```

4. 二つのリスト [a1; ...; an] と [b1; ...; bn] を引数として, [(a1, b1); ...; (an, bn)] を返す関数 zip. (リストの長さが異なる場合は長いリストの余った部分を捨ててよい.)

Exercise 5.5 f, g を適当な型の関数とする．map f (map g l) を map を一度しか使用しない同じ意味の式に書き換えよ．map (fun x -> ...) l の ... 部分は?

Exercise 5.6 forall, exists を fold_right, map を組み合わせて定義せよ.

Exercise 5.7 quick 関数を @ を使わないように書き換える．quicker は未ソートのリスト l と，sorted というソート済でその要素の最小値が l の要素の最大値より大きいようなリストを受け取る．定義を完成させよ.

```
let rec quicker l sorted = ...
```

A　レポートその 2: 締切 11 月 27 日

必修課題:　4.1, 4.2, 4.5, 4,6, 4,8, 5.1, 5.4 から 3 つ，5.5, 5.7

オプション課題:　第 4 回，第 5 回の資料の残り全部の問題.