順列生成

問題
リストを与えると、その要素のすべての並び方を返す関数 junretu を作りなさい。(要素に重複はないものとする)
例: (junretu '(a b c)) → ((a b c) (a c b) (b c a) (b a c) (c b a) (c a b))

解法の一例
リストのある要素に対して，残りの要素の順列並びの先頭に付け加える（この部分は再帰)。この操作を元のリストの全要素に対して行い、結果をくっつける。
たとえば、リスト (a b c) に対しては、a を残りの順列並びの結果 ((b c) (c b)) のそれぞれの先頭に加えて、((a b c) (a c b)) を得る。b に対しては ((a c) (c a)) に加えて、((b a c) (b c a)) を得る。c に対しては ((a b) (b a)) に加えて、((c a b) (c b a)) を得る。この 3 つの結果を合わせて、最終結果が得られる。

いま、ためしに

(defun abc (x) (mapcar #'(lambda (e) (cons e (remove e x))) x))

というものを作ると、

(abc '(a b c)) → ((a b c) (b a c) (c a b))

となり、'ある要素' と 'その要素を除いたもの' のリストを並べたものができる。

そこで、'その要素を除いたもの' の代わりに 'その要素を除いたもののすべての並び' にすればよいのではないかと考えを進める。'その要素を除いたもののすべての並び' というのは、まさに 'その要素を除いたものの順列' なので、

```
(defun efg (x) (mapcar #'(lambda (e) (cons e (efg (remove e x))))
                       x))
```

または

```
(defun efg (x)
   (if (null x) nil
             (mapcar #'(lambda (e) (cons e (efg (remove e x))))
                     x)))
```

としてみると、

(efg '(a b c)) → ((a (b (c)) (c (b))) (b (a (c)) (c (a))) (c (a (b)) (b (a))))

となり、再帰の結果として得られるすべての順列に対して、最初の要素を加えていないことがわかる。これを正すには、

```
(defun efg (x)
  (if (null x) nil
        (mapcar #'(lambda (e)
            (mapcar #'(lambda (p) (cons e p))
                (efg (remove e x))))
         x)))
```

のようにすればいいと思われるが、これだと再帰のストッパーの返値がダメで、要素が消えてしまう。そこで、

```
(defun efg (x)
  (if (null x) '(())
      (mapcar #'(lambda (e)
          (mapcar #'(lambda (p) (cons e p))
              (efg (remove e x))))
       x)))
```

とすると、

(efg '(a b c)) → (((a (b (c))) (a (c (b)))) ((b (a (c))) (b (c (a)))) ((c (a (b))) (c (b (a)))))

となり、括弧を無視すれば順列となっていることがわかる。余分な括弧をなくすには、いつかの授業で取り上げた 関数 reduce を使って、下のような解が導ける。（授業でやった関数だけで出来ています！）

```
(defun junretu (x)
  (if (null x) '(())
      (reduce #'append
         (mapcar #'(lambda (e)
            (mapcar #'(lambda (p) (cons e p))
                (junretu (remove e x))))
          x))))
```

別解 1

余分な括弧をなくすには、外側の mapcar を変えて、合わせながら append するという考えで、次のような mapappend 関数を導入してもいいでしょう。

(mapappend #'(lambda (x) x) '(((a b c) (d e f)) ((g h)))) → ((a b c) (d e f) (g h))

```
(defun junretu (x)
  (if (null x) '(())
      (mapappend #'(lambda (e)
          (mapcar #'(lambda (p) (cons e p))
              (junretu (remove e x))))
       x)))

(defun mapappend (func x)
  (if (null x) nil
      (append (funcall func (car x)) (mapappend func (cdr x)))))
```

別解 2

実は mapcan という mapappend と同様の仕事をしてくれる（破壊的な）関数が存在するので、それを使ってもよい。

(mapcan #'(lambda (x) x) '(((a b c) (d e f)) ((g h)))) → ((a b c) (d e f) (g h))

見た目が美しい解答例です。

```
(defun junretu (x)
  (if (null x) '(())
      (mapcan #'(lambda (e)
          (mapcar #'(lambda (p) (cons e p))
              (junretu (remove e x))))
       x)))
```

いずれの解答例も、再帰の表現力を発揮した短くも奥が深いプログラムとなっています。