プログラミング入門
第 11 回「可変データ応用」

二宮 崇（ ninomiya@cs.ehime-u.ac.jp ）

教科書

Structure and Interpretation of Computer Programs, 2nd Edition: Harold Abelson, Gerald Jay Sussman, Julie Sussman, The MIT Press, 1996

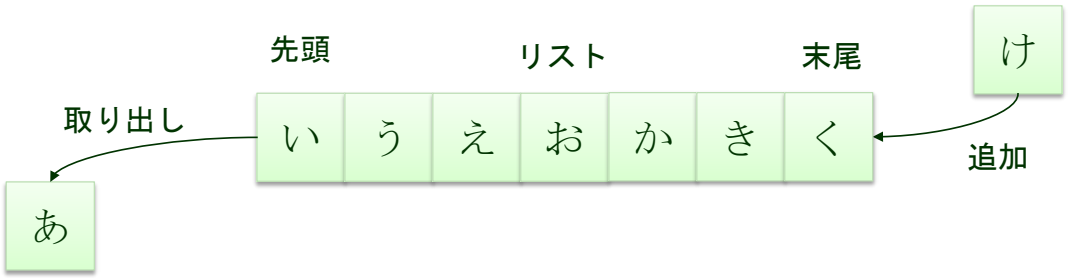
CC BY NC

標準部品化力、オブジェクトおよび状態（第 3 章）

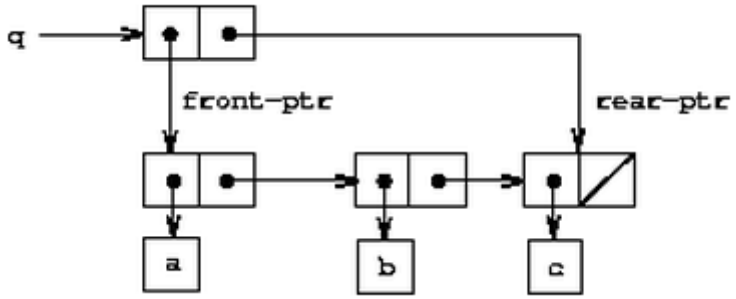
3.3 可変データのモデル化

3.3.2 キューの表現

set-car!やset-cdr!を用いることで、キューを作成することができるようになります。次々とやってくるデータを次々と追加し、必要に応じて追加したデータを取り出せるリストがあると便利ですが、追加するときは末尾に追加し、取り出すときは必ず先頭から取り出すようにしたリストのことをキューと呼びます。キューは、最初に入れたものが最初にでてくるので、FIFO (first in first out)とも呼ばれます。



キューに効率を良くするために、先頭と、末尾のデータを指すポインタを持つようにします。



まず、基本的な関数です。

```
(define (make-queue) (cons '() '()))

(define (front-ptr queue) (car queue))

(define (rear-ptr queue) (cdr queue))

(define (set-front-ptr! queue item) (set-car! queue item))

(define (set-rear-ptr! queue item) (set-cdr! queue item))
```

このように set-car!や set-cdr!を使うと、関数を返すことによってオブジェクトを作成することに比べかなり容易にデータオブジェクトがつくれる、ということがわかります。

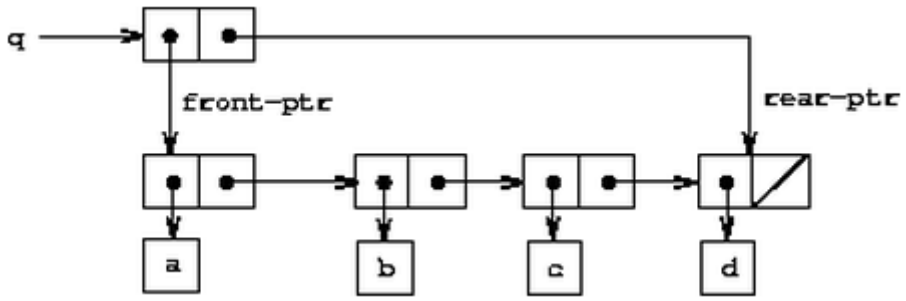
　続いて、空であるかどうかを判定するためには、front-ptr が空であるかどうかを判定すれば良いので、

```
(define (empty-queue? queue) (null? (front-ptr queue)))
```

となって、先頭の要素を見るためには、front-ptr が指すリストの先頭の要素を返せば良い、ということになります。

```
(define (front-queue queue)
  (if (empty-queue? queue)
      (error "FRONT called with an empty queue" queue)
      (car (front-ptr queue))))
```

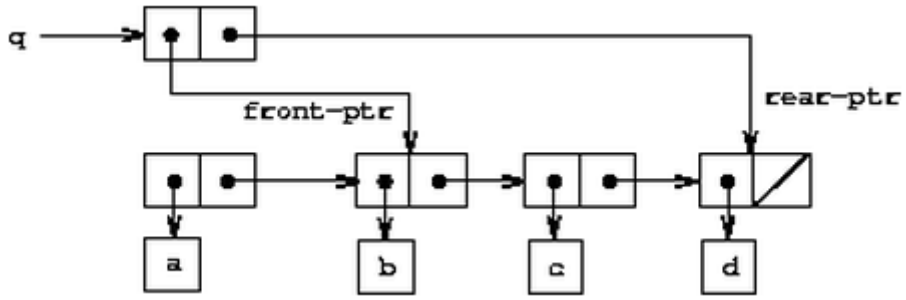
次に、挿入ですが、キューは常に末尾にデータが加わるので、set-rear-ptr!で末尾に新しい要素を追加してやれば良い、ということになります。例えば、p.1 の図に(insert-queue! q 'd)を実行すると次のような状態になります。



また、キューが空である場合には特殊な操作が必要で、front-ptr と rear-ptr が同じ新しいアイテムを指すようにする必要があります。

```
(define (insert-queue! queue item)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair)
           queue)
          (else
           (set-cdr! (rear-ptr queue) new-pair)
           (set-rear-ptr! queue new-pair)
           queue))))
```

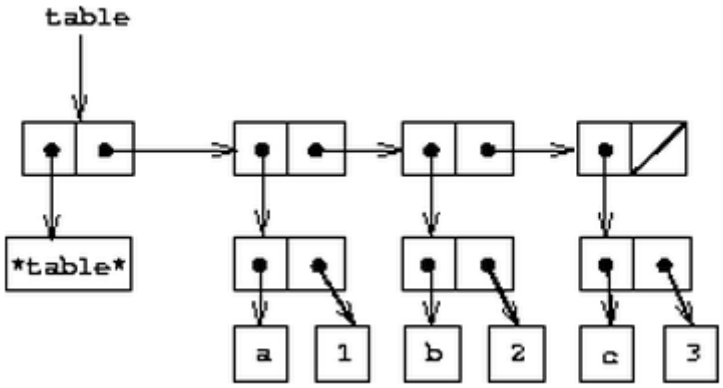
最後に delete は、先頭の要素を削除すれば良いので、set-front-ptr!で、front-ptr を一つ後ろにつなげなおせば ok です。例えば、p.2 の図のキューに(delete-queue! q)を実行すると次のような状態になります。




```
(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error "DELETE! called with an empty queue" queue))
        (else
         (set-front-ptr! queue (cdr (front-ptr queue)))
         queue)))
```

3.3.3 表の表現

次に連想リスト(キーと値による１次元の表)のオブジェクトをつくることを考えてみます。連想リストが空になったときに困らないように、ダミーのエントリーもつくっておきます。



ここでは、*table*とあるのがダミーのエントリーで、table 全体はリストになっています。先頭の要素がダミーで、次の要素から、キーと値のペアが格納されている、とい

うことになります。このようなリストは頭つきリスト(headed list)と呼ばれ、*table*をもつペアが頭(head)で、それに続くリストは背骨(backbone)と呼ばれます。上の table は a: 1, b: 2, c: 3 という連想リストを表現しています。

ダミーは連想リストが空になったときや、リストの先頭の要素を削除したときに困らないようにするために用意されていますが、ある意味、ダミーのエントリーが連想リストを表すデータ構造の本体、つまり連想リストオブジェクトと考えることもできます。このようなダミーがないといろいろと不便が生じます。例えば、仮に table が上の((a.1) (b.2) (c.3))を指していたとします。ここで、(a.1)を削除しようとしたら、どうなるでしょう？プログラムを書いてみるとわかりやすいのですが、table という変数を set!で書き換えないといけないのですが、連想リストを指す変数名は table だけとは限らないので、単に(set! table (cdr table))とすればいいわけではありません。つまり、どんな名前の連想リストに対してもその連想リストを操作できる関数を記述しなければいけないのですが、それは非常に難しいです(Scheme ではおそらくできない)。上記のダミーがこのような問題を解決するのに大変有効と言えます。

このようなデータに対する検索は次の lookup で実現できます。

```
(define (myassoc key records)
  (cond ((null? records) #f)
        ((equal? key (car (car records))) (car records))
        (else (myassoc key (cdr records)))))
(define (lookup key table)
  (let ((record (myassoc key (cdr table))))
    (if record
        (cdr record)
        #f)))
```

等しいキーがあるかどうか順に探していきます。

新しいエントリーを挿入するプログラムは次のようになります。

```
(define (insert! key value table)
```

```
  (let ((record (myassoc key (cdr table))))
    (if record
        (set-cdr! record value)
        (set-cdr! table
                  (cons (cons key value) (cdr table)))))
  'ok)
```

最後に、新しい連想リストを作るプログラムは次のようになります。

```
(define (make-table) (list '*table*))
```

実行例は次のようになります。

```
> (define a (make-table))
> a
(*table*)
> (insert! 'a 1 a)
ok
> a
(*table* (a . 1))
> (insert! 'b 2 a)
ok
> (insert! 'c 3 a)
ok
> (lookup 'b a)
2
> a
```

```
(*table* (c . 3) (b . 2) (a . 1))
```

クォートによるリストと LIST 手続きの違い

クォートによるリストと list 関数によって得られるリストの違いを認識していないと問題になることがあります。

クォートによって得られるリスト: '(a b c)

list 関数によって得られるリスト: (list 'a 'b 'c)

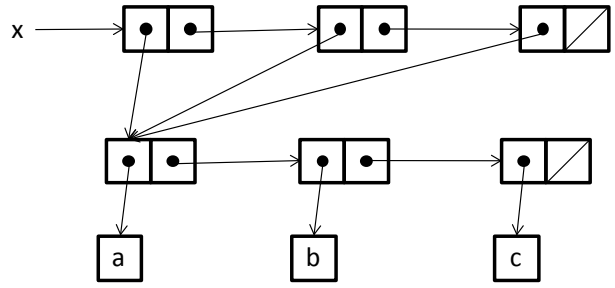
どちらも同じものを返すようにみえますが、実は大きな違いがあります。クォートによって作られるリストはプログラム上にただ１カ所にしか存在せず、そのポインタが返されます。一方 list 関数は、そこに新しいリスト構造を生成する、ということを行います。例えば、次のプログラムと実行例をみてください。

```
(define (f) '(a b c))

(define (g) (list 'a 'b 'c))

(define x (list (f) (f) (f)))

(define y (list (g) (g) (g)))

> x

((a b c) (a b c) (a b c))

> y

((a b c) (a b c) (a b c))

> (set-car! (car x) 'change)

> (set-car! (car y) 'change)

> x

((change b c) (change b c) (change b c))

> y
```

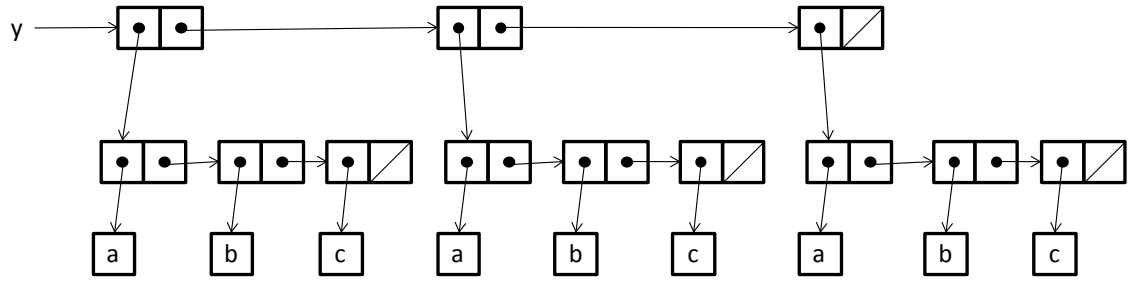
((change b c) (a b c) (a b c))

クォートで作成した x のほうは全ての'a が'change に置き換わったのに、list 関数で作成した y のほうは先頭の'a だけが'change に置き換わっています。

これは、上記のクォートで生成された x は次のようなデータ構造になっていて、



list 手続きによって生成された y は次のようなデータ構造になっているためです。



さらにここで f を実行してみます。

> (f)

(change b c)

f という関数の中のリストも変化していることがわかります。

この現象を理解するのはちょっと難しいですが、次のように考えれば良いと思います。まず、プログラムをインタプリタが読み込みます。プログラムを読み込むとき(define (f x)…)といった関数の定義を読みこむと、そのときに関数を表す関数オブジェクトが生成されます。その関数の中にクォートによるリストが記述されてあった場合はクォートによるリストは関数と一緒に生成されて、関数の中に埋め込まれます。よって、なんらかの関数が返すクォートによるリストは同一のリストが返されるようになります。一方、list 関数によるリストは、プログラム読み込み時には、まだリストが生成されていません。インタプリタ上で、このプログラムを実行させ、その list 関数が実行され

た時に初めてリストオブジェクトが生成されます。また、このとき同一のリストオブジェクトを返さず、新しいリストオブジェクトを動的に生成します。

一言で言うと、クォートによるリストは静的に生成されるリストで、list 関数によるリストは動的に生成されるリストということもできます。