プログラミング入門
第 10 回「可変リスト」

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

第 3 章 標準部品化力、オブジェクトおよび状態

3.1.2 代入を取り入れた利点

代入を扱えることで、手続き型言語のような便利さも得られるようになります。また、乱数を発生する手続きの実現や、それをつかったモンテカルロシミュレーションなどもより容易に実現できるようになります。何より、オブジェクト指向型のプログラミングが可能となります。

3.1.3 代入を取り入れた代価

代入によって、手続き型言語で得られていた便利さや、オブジェクト指向型言語で得られていた利点が得られるようになります。しかし、その代償もまた大きいものとなります。まず、この計算モデルは置き換えモデルで解釈できなくなっています。また、「素敵な」数学的性質をもった単純なモデルでこれらの代入を扱えるものはありません。今までの関数は、同じ引数なら必ず同じ結果が返ってきていたので、数学的な関数とみなすことができました。従って、今までの代入を用いない関数によるプログラミングは関数型プログラミングと呼ばれます。

代入による代償は次のようになります。

置き換えモデルを使って解釈することができない

例えば、

```
(define (f i)
   (set! i (+ i 1))
   i)
```

という、簡単な関数 f を考えるとします。これに対しては、

```
> (f 10)
11
```

となりますが、置き換えモデルを用いると、

```
(f 10)
=(set! i (+ 10 1)) 10    ;式が二つ並んでいることに注意
=10
```

となり、i を 10+1 に set!したあと、10 を返すということになってしまいます（本当なら、10+1 の結果である 11 を返したい）。

同一と変化

式の評価の結果を変えることなく、式の中で「等しいものは等しいもので置き換えられる」という概念の成り立つ言語を参照透明（referentially transparent)であるといいます。set!を導入した時点でこの参照透明性がなくなってしまいます。これは計算モデルを設計する際に、きれいなモデルによる構築が非常に難しくなってしまうことを意味します。

また、これは、計算オブジェクトが同一であるかどうか、ということを形式的に判定することが難しくなってしまった、ということも意味します。

例えば、次の例をみてみましょう。

```
(define peter-acc (make-account 100))
```

```
(define paul-acc (make-account 100))
```

とモデル化するのと、

```
(define peter-acc (make-account 100))

(define paul-acc peter-acc)
```

とモデル化するのとでは、本質的に異なることをやっています。(上は異なる二つのオブジェクトができているのに対し、下では、一つのオブジェクトを共有していることになります)。set!がもし使われなかったとしたら、上の定義も下の定義も同一になって、peter-acc も paul-acc も同一であることがすぐにわかります。しかし、set!を使うと、上の定義と下の定義では本質的に異なることをやっていることに注意をしなくてはいけなくなります。

手続き型プログラムの落とし穴

手続き型の流儀は計算モデルの複雑性を高める上に、関数型プログラミングでは起こりえないバグが入りえます。

まず、関数型プログラミングによる階乗の計算をみてみましょう。

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

次にこれに対して、set!を使った階乗の計算をみてみましょう。

```
(define (factorial n)
```

```
  (let ((product 1)
        (counter 1))
    (define (iter)
      (if (> counter n)
          product
          (begin (set! product (* counter product))
                 (set! counter (+ counter 1))
                 (iter))))
    (iter)))
```

しかし、代入の順番を逆にして、

```
(set! counter (+ counter 1))
(set! product (* counter product))
```

とすると、まったく正しくない結果となってしまいます。関数型プログラミングではこういう気遣いは必要ありませんでした。ただし、for と range を組み合わせれば手続き型プログラミングでも比較的バグのでにくいプログラムを記述できるようになります。次のプログラムは for と range を使った factorial です。

```
(define (factorial n)
  (define product 1)
  (for (range 1 n)
   (lambda (i) (set! product (* i product))))
  product)
```

3.3 可変データのモデル化

一度束縛された変数を変更するには set!で変更できることがわかりました。例えば、口座の金額を変更することを学びました。しかし、複雑な変更を行う関数だけあれば良いというわけではなく、データ抽象の考えに沿って、データ構造の作成、読み込みといった基本関数があれば他の複雑な関数も基本関数から作成することが可能となります。さらに、今、データの変更が可能なオブジェクト(可変データオブジェクト, mutable data object)を扱っているので、データ構造への変更も基本関数としてあると大変便利になります。たとえば、口座の場合は、口座をつくる make-account, 口座の残高を照会する inquiry の他に、残高を変更する(set-balance! <account> <new-value>)といったものがあれば、これらの機能を使って、より複雑な関数を記述することができるようになります。

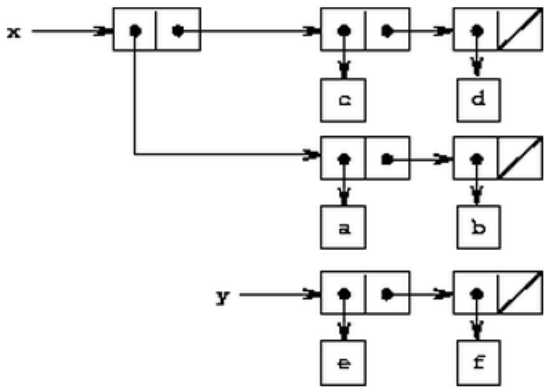
この節では、ペアに対する変更関数を学びます。これによって、リスト、木構造といったデータの上書きができるようになります。

3.3.1 可変リスト構造

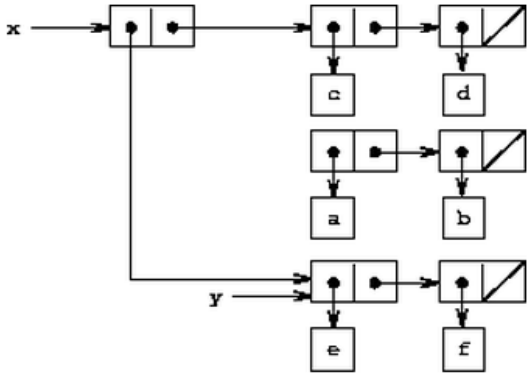
ペアの基本変更子は、set-car!と set-cdr!になります。

(set-car! ⟨変更対象の対⟩ ⟨変更内容⟩)

変更対象の対の car のポインタを変更内容につなげかえます。例えば、次の図のように x がリスト'((a b) c d)で、y がリスト'(e f)に束縛されているとします。

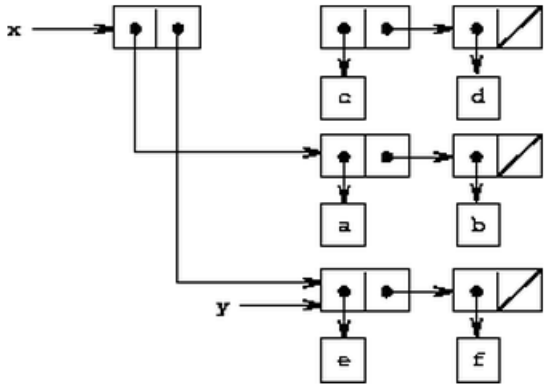


(set-car! x y)を実行すると、次の図のように変更されます。



ここで注意することは、元々あった(a b)というリストはどこからも参照されなくなった、ということと、x の car ポインタは y を指すようになった、ということです。

次に set-cdr!についてですが、set-cdr!は set-car!に似ていて、car のポインタを変更するのではなく、cdr のポインタを変更します。例えば、元々の x と y に対して、(set-cdr! x y)を実行すると次のようになります。



このように、set-car!と set-cdr!を用いると、ペアの構造を変更することができ、キューを作ったり、効率の良いリスト操作の関数を実現することが出来るようになります。ただし、この操作は副作用を伴う変更なので、注意が必要です。例えば、次の append の例をみてみましょう。set-car!や set-cdr!を使わない場合は、

```
> (define x '(a b c d))

> (define y '(e f))

> (define z (append x y))

> x
```

```
(a b c d)
> y
(e f)
> z
(a b c d e f)
```

となって、元々のｘやｙに変化がありませんが、set-car!やset-cdr!を使う場合、例えば、

```
(define (last-pair x)
  (if (null? (cdr x))
      x
      (last-pair (cdr x))))

(define (append! x y)
  (set-cdr! (last-pair x) y)
  x)
```

とプログラムを書いた場合、

```
> (define x '(a b c d))
> (define y '(e f))
> x
(a b c d)
> y
(e f)
> (append! x y)
```

(a b c d e f)

> x

(a b c d e f)

> y

(e f)

となって、元々の x が書き換えられていることに注意してください。また、よくある勘違いとしては次のようなものがあります。

> (define x '(a b c d))

> (set! (car x) 3)

set!: not an identifier in: (car x)

set-car!の代わりに set!で書き換えたくなるのですが、set!は define と同様に特殊形式で変数だけを引数としてとるためうまく処理されません。それで、次のように

> (define x '(a b c d))

> (define y (car x))

> (set! y 3)

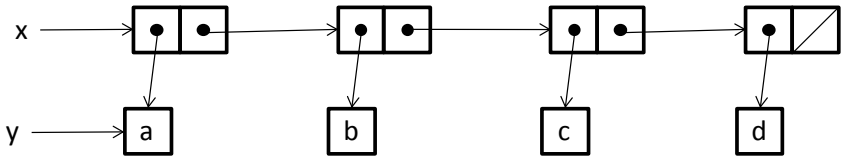
> x

(a b c d)

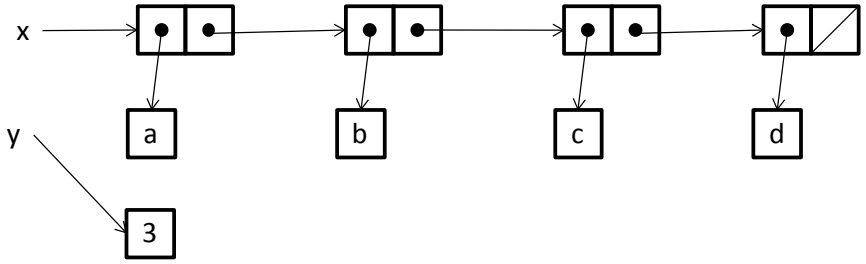
> y

3

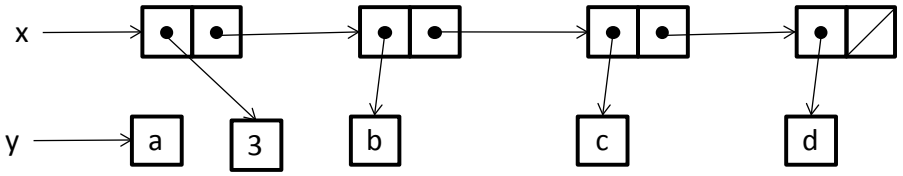
とやって、x の car をとって、y としたあと、y を書き換えれば、set-car!と同じことができるような気がするのですが、やはりだめで、x は元のままです。かわりに y の値だけが変わっています。set!を実行する前は次のようになっていますが、



(set! y 3)を実行すると次のようになります。



つまり、y が指している箱が 3 の箱に置き換わった、ということです。set!はポインタが指しているデータを直接書き換えるのではなく、変数の指しているポインタを書き換えていることに注意が必要です。結局、x の先頭の要素を書き換えるには set-car!が必要になります。(set-car! x 3)とした場合は、次のような結果になりますが、逆に y は元のままになっていることに注意してください。



この違いを理解するのは簡単なようで意外と難しく、これが原因でバグがよく発生します。基本的には、ポインタと値を明示的に区別していない言語では、ポインタのみを書き換えていて、値を直接書き換えることを行いません。何故値を直接書き換えられないのか、ということを不思議に思うかもしれませんが、インタプリタやコンパイラの設計や効率の問題が関係してきて簡単には実現できない、ということで理解してください。set!や set-car!、set-cdr!を使えるようになって便利になった代わりに、箱とポインタ表現を使って値とポインタを区別しながら理解しないと何が起きているのか全く理解できないケースがでてくるようになったということです。

代入操作に必要な基本的な手続きについては、set!, set-car!, set-cdr!で全てになります。これらを組み合わせて用いれば、どんな複雑なデータ構造に対してもその一部だけ書き換えをするといったことができるようになります。

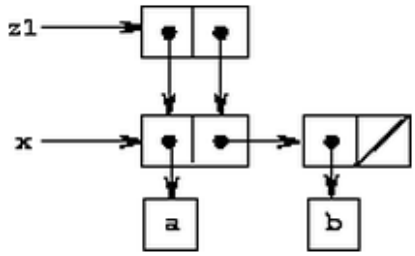
共有と同一

set-car!や set-cdr!で変更する場合に、そのデータ構造によって、結果が大きく異なってくることに注意しよう。例えば、

(define x (list 'a 'b))

(define z1 (cons x x))

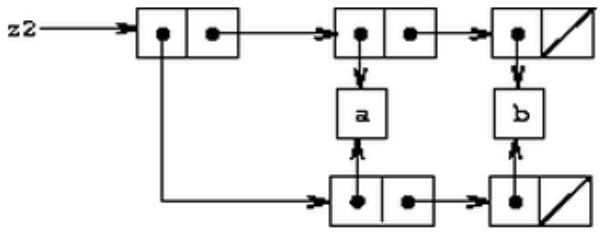
とやって、データを作成すると、次のようなデータ構造になります。



続いて、

(define z2 (cons (list 'a 'b) (list 'a 'b)))

として z2 を作成すると、次のようなデータ構造が得られます。



インタプリタで表示させた場合は、

> x

(a b)

> z1

((a b) a b)

> z2

((a b) a b)

となって、同じ表示になるのですが、そのデータ構造が異なるため、set-car!やset-cdr!の結果が異なる場合がでてきます。例えば、

> (set-car! (car z1) 'wow)

> z1

((wow b) wow b)

> (set-car! (car z2) 'wow)

> z2

((wow b) a b)

という異なる結果になります。つまり、データオブジェクトがどうデータを共有しているか注意しないと、思いもしない結果になりうる、ということです。

　データが共有されているかどうかということは非常に重要なことなので、これを判定する関数が必要です。実は、今まで同じシンボルかどうかを判定してきたeq?がそれを判定してくれます。(eq? x y)でxとyが同じオブジェクトを指していれば(つまり、xとyがポインタとして等しければ)真が返ってきます。シンボルや整数は(実装はさておき)概念的にはどこで出てきても同じデータを指していることになっています。

> (eq? 1 1)

#t

> (eq? 1.0 1.0)

#f

> (eq? 'a 'a)

#t

> (eq? (car z1) (cdr z1))

#t

> (eq? (car z2) (cdr z2))

#f

変更は単なる代入

ペアは次のように関数で実現できました。

```
(define (mycons x y)
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          (else (error "Undefined operation -- CONS" m))))
  dispatch)
(define (mycar z) (z 'car))
(define (mycdr z) (z 'cdr))
```

set-car!や set-cdr!も同様に set!を使って実現することができます。

```
(define (mycons x y)
  (define (set-x! v) (set! x v))
  (define (set-y! v) (set! y v))
  (define (dispatch m)
    (cond ((eq? m 'car) x)
          ((eq? m 'cdr) y)
          ((eq? m 'set-car!) set-x!)
          ((eq? m 'set-cdr!) set-y!)
          (else (error "Undefined operation -- CONS" m))))
```

```
  dispatch)
(define (mycar z) (z 'car))
(define (mycdr z) (z 'cdr))
(define (my-set-car! z new-value)
  ((z 'set-car!) new-value)
  z)
(define (my-set-cdr! z new-value)
  ((z 'set-cdr!) new-value)
  z)
```

上記の例からわかるように、set-car!や set-cdr!は set!があれば cons, car, cdr と同様に自分で実装できるものだったということがわかります。