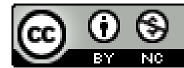
プログラミング入門
第６回「データ構造とリスト」

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



2. データによる抽象の構築

1 章では単純な数値データのみ扱ってきました。しかし、様々な実世界のデータをプログラムで処理するためには、数値だけではなくより複雑なデータ構造を用いることが必要となります。第 2 章ではデータオブジェクトを組み合わせ、合成データを作って抽象化を構築する機能について勉強していきます。

2.1 データ抽象入門

1 章では関数に名前をつけ、細部の実装を隠すことにより、関数の抽象化が行われてきました。同様に合成データに対する同様の考え方をデータ抽象と呼びます。

ペア (pair)

cons: ペアを生成する関数

car: ペアの左側の要素(第 1 の要素)を取り出す関数

cdr: ペアの右側の要素(第 2 の要素)を取り出す関数

```
(define x (cons 1 2))
(car x)  →  1
(cdr x)  →  2
```

ペアが要素のペアも作れます。

```
(define x (cons 1 2))
(define y (cons 3 4))
(define z (cons x y))
(define a (cons (cons 1 2) (cons (cons 3 4) (cons 5 6))))
```

2.1.1 有理数の算術演算

次のペアを用いることで例えば有理数を表現することができます。

```
(define (make-rat n d) (cons n d))
(define (numer x) (car x))
(define (denom x) (cdr x))
(define (print-rat x)
  (newline)
  (display (numer x))
  (display "/")
  (display (denom x)))
```

有理数の足し算は次のように定義できます。

```
(define (add-rat x y)
```

```
  (make-rat (+ (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
```

同様に引き算、掛け算、割り算も定義することができます。

```
(define (sub-rat x y)
  (make-rat (- (* (numer x) (denom y))
               (* (numer y) (denom x)))
            (* (denom x) (denom y))))
(define (mul-rat x y)
  (make-rat (* (numer x) (numer y))
            (* (denom x) (denom y))))
(define (div-rat x y)
  (make-rat (* (numer x) (denom y))
            (* (denom x) (numer y))))
```

ここで重要なことは、有理数に対する関数の定義を与えるときに、cons や car, cdr をつかっていないことに注目してください。合成データに対し、cons, car, cdr で直接生成、アクセスするのではなく、make-rat, numer, denom で生成、アクセスするようにデータ抽象化がなされています。有理数という合成データが具体的にどのように作られているか気にすることなく、有理数の関数を定義できる、ということです。こういう抽象化によって、直感的にわかりやすい操作が行えるということ、また、データに対するインターフェースが揃っていることによって、インターフェースレベルでのプログラムの改良が行える、ということです。例えば、make-rat を分子と分母で簡約するようにするならば、

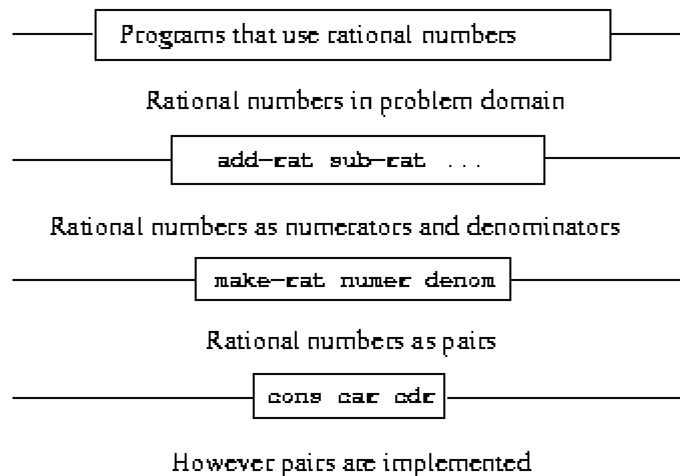
```
 (define (make-rat n d)
```

```
  (let ((g (gcd n d)))
    (cons (/ n g) (/ d g))))
```

と書き換えれば、有理数を生成する全ての場所で、必ず簡約されることになります(gcd は最大公約数を返す racket の組み込み関数)。これが、make-rat を使わずに直接 cons で有理数を生成するようになっていると、プログラムの全ての cons を書き換えなければならない上に、各々の cons が、有理数のための cons なのか、別のデータ、例えば、複素数やリストなど、なのかいちいち判別して、プログラムを書き換えていかなければならなくなります。

2.1.2 抽象の壁

有理数の例では、次のようなレベルが存在しています。



add-rat や sub-rat を実装するときは、con や car を気にすることなく、その下のレベルの make-rat, numer, denom で実装することができる。さらに上のレベルからは、add-rat や sub-rat を使って有理数の操作ができる。上の図の水平線が、抽象化のレベルの壁を表現しています。

例えば、make-rat の時点で簡約かするのではなく、値を取り出したいときに簡約化するように変更することが次のようにできます。

```
(define (make-rat n d) (cons n d))
```

```
 (define (numer x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (car x) g)))
(define (denom x)
  (let ((g (gcd (car x) (cdr x))))
    (/ (cdr x) g)))
```

このように変更したとしてもその上のレベルのプログラムはまったく書き換える必要がないのです！

2.1.3 データとは何か

cons で作られるデータは、実際には racket の組み込みデータとして、直接実装されています。しかし、cons, car, cdr は今まで 1 章で習ってきた関数で実現することができます。

```
(define (my-cons x y)
  (define (dispatch m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "Argument not 0 or 1 -- CONS" m))))
  dispatch)
(define (my-car z) (z 0))
(define (my-cdr z) (z 1))
```

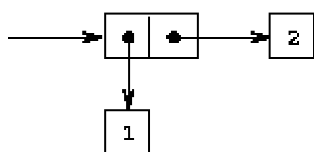
この例において、関数 dispatch を返すとき、関数の中に出現する自由変数 x, y の値については、関数の中に記憶されたまま関数と一緒に返される、ということを示しています。つまり、関数は変数とその値を記憶する能力をもっているということです。実際にこのように cons が実装されているわけではないのですが、cons をつかったデータ構造

は全て関数に書き換えられる、ということが重要です。また、Scheme では全てのデータ構造は cons, car, cdr をつかって記述されるので、全てのデータ構造は全て関数に書き換えられる、ということです。

2.2 階層データ構造と閉包性

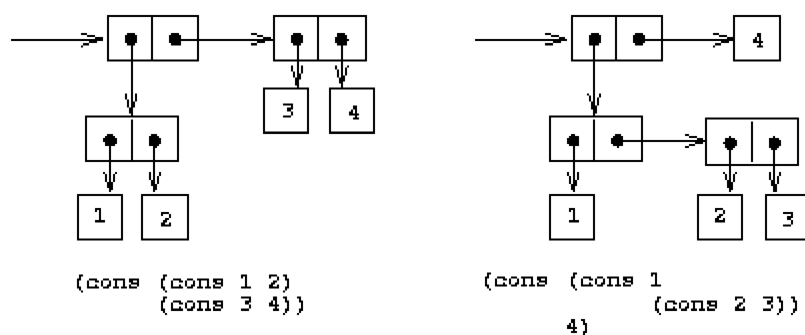
箱とポインタ記法

cons で作ったペアを視覚化するために次のような記法を使います。



これは cons(1, 2) を表します。この記法は箱とポインタ記法 (box-and-pointer notation) と呼ばれます。ペアは二つの箱で表現されていて、左側に car で取り出されるデータへのポインタが、右側には cdr で取り出されるデータへのポインタが格納されています。

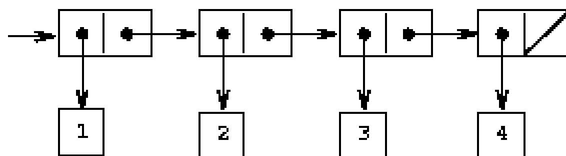
ペアの要素にペアを格納することができるので、1, 2, 3, 4 という数値を次のように組み合わせて表現することができます。



要素がペアであるようなペアを作る能力は非常に重要で、cons の閉包性と呼ばれます。一般に、データオブジェクトを組み合わせる演算は、その演算をつかって何かを組み合わせた結果がまた同じ演算をつかって組み合わせられるという時、閉包性を満足する、といいます。

2.2.1 系列の表現

ペアをつかって系列(sequence)を表現することができます。次の例は 1,2,3,4 という系列を表現しています。



みてわかるように、car に対応するポインタが系列の要素を指していて、cdr に対応するポインタが次の系列を指しています。最後の斜線は系列の終端を表します。これは、

```
(cons 1 (cons 2 (cons 3 (cons 4 '()))))
```

として作られます。このように入れ子の cons で作られた系列のことをリストと呼びます。Scheme にはリストを作る基本的関数が用意されています。一般に、

(list $\langle a_1 \rangle$ $\langle a_2 \rangle$ … $\langle a_n \rangle$)

は

(cons $\langle a_1 \rangle$ (cons $\langle a_2 \rangle$ (cons … (cons $\langle a_n \rangle$ '())…)))

と等価です。また、リストは表示されるときに、(1 2 3 4)という風に見やすくなるように表示されます。

リストがこのようなデータ構造になっているので、car はリストの最初の要素を取り出し、cdr は先頭を除いた部分リストを取り出すことになることに注意しましょう。

終端は'()という特殊な記号で表現され、これは、空リスト (empty list)と考えれば良いです。Scheme の実装によっては、empty、nil、null も使えたりします。

次の操作をやってみましょう。

```
(define one-through-four (list 1 2 3 4))

(car one-through-four)

(cdr one-through-four)
```

```
(car (cdr one-through-four))

(cons 10 one-through-four)

(cons 5 one-through-four)
```

リスト演算

次の関数はn+1番目の要素を取り出します。(リストは0から番号をつける習慣がある)

```
(define (list-ref items n)
  (if (= n 0)
      (car items)
      (list-ref (cdr items) (- n 1))))
```

例: (list-ref (list 1 4 9 16 25) 2) → 9

リストには次の基本的関数が用意されています。

null?: 引数が空リストであるかどうかを判定する関数

これをつかってリストの長さを返す関数を作れます。

```
(define (length items)
  (if (null? items)
      0
      (+ 1 (length (cdr items)))))
```

例: (length (list 1 4 9 16 25)) → 5

次にリストをつなげる関数を定義します。

```
(define (append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1) (append (cdr list1) list2))))
```

例: (append (list 1 2 3) (list 4 5 6 7 8)) → (1 2 3 4 5 6 7 8)