システム理論演習 II

(4) 抽象データ・階層データ

増原英彦 *(masuhara@graco.c.u-tokyo.ac.jp)*

1997 年 5 月 9 日

1　抽象データ

複合データ (compound data) とは、複数のデータを 1 つのデータとしてまとめあげる手段である。(*cf.* C の `struct`, Pascal の `record`.)

例えば分数を扱うプログラムを考える。2 つの分数の和の計算するには、1 つ目の分数の分母と分子、2 つ目の分数の分母と分子をもとに、和の分数と分母と分子を求めてあげればよい......が、1 つの分数を 1 つのデータとして扱えた方が便利である。つまり、1 つの分数 = 分母と分子の組、としたい。

1.1　データの抽象化

- データが実際にはどのようにようにして作られているか、を意識する世界を内部表現という
- データを使用する世界では、構築子 (constructor) を使って複合データを作り、選択子 (selector) を使って、データの要素をとり出す

この二つの世界を独立させて扱うのが、データの抽象化の基本である。

例えば、分数の例では、以下のような関数が構築子・選択子となり得る。

`(make-rat` $\langle n\rangle$ $\langle d\rangle$`)`: 分子が $\langle n\rangle$、分母が $\langle d\rangle$ のような分数を作る

`(numer` $\langle x\rangle$`)`: 分数 $\langle x\rangle$ の分子を返す

`(denom` $\langle x\rangle$`)` 分数 $\langle x\rangle$ の分母を返す

内部表現を後回しにして、まずは分数を使う世界として、四則演算などを定義する。

分母の加減乗除および同値性は、

$$
\begin{aligned}
\frac{n_1}{d_1}+\frac{n_2}{d_2} &= \frac{n_1d_2+n_2d_1}{d_1d_2} \\
\frac{n_1}{d_1}-\frac{n_2}{d_2} &= \frac{n_1d_2-n_2d_1}{d_1d_2} \\
\frac{n_1}{d_1}\cdot\frac{n_2}{d_2} &= \frac{n_1n_2}{d_1d_2} \\
\frac{n_1/d_1}{n_2/d_2} &= \frac{n_1d_2}{d_1n_2} \\
\frac{n_1}{d_1} &= \frac{n_2}{d_2} \qquad n_1=n_2\text{かつ}d_1=d_2\text{の場合}
\end{aligned}
$$

のような定義なので、

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

のように定義できる。

さらに、分数を印字するための手続き

```
(define (print-rat x)
  (display (numer x))
  (display "/")
  (display (denom x))
  (newline))
```

を定義しておくと便利である。

課題 4-1: (分数の計算)

add-rat の定義にならって、引き算・掛け算・割り算と、同値性を調べる関数 sub-rat mul-rat div-rat equal-rat? を定義せよ。定義は、例えば rat-simple.scm という名前で一つのファイルにしておくと後々便利である。 ∎

1.2 対 (pair)

複合データの内部表現を考える。 Scheme には、複合データを作るために対 (pair) というものがあり、そのために次のような原始的な関数が用意されている。

cons: 2 つの引数を合成して、 1 つのデータとして返す

car: 合成されたデータから 1 つ目の値をとり出す

cdr: 合成されたデータから 2 つ目の値をとり出す

これらの関数を使うことで、複数のデータを一つの値として扱うことができる。

```
> (define x (cons 1 2))    ; 1 と 2 の対を作る
> (car x)                  ; 対の 1 番目
1
> (cdr x)                  ; 対の 2 番目
2
> (define y (cons 3 x))    ; 3 と x の対を作る
> (car y)                  ; 対の 1 番目
3
> (car (cdr y))            ; 対の 2 番目 (つまり対 x) の 1 番目
1
```

この対を使って、分数の内部表現を定義すると、

```
(define (make-rat n d) (cons n d))
(define (numer x) (car x))
(define (denom x) (cdr x))
```

のようになる。

このようにして定義した分数は、一個のデータとして扱うことができる。

```
> (define one-half (make-rat 1 2))
> (print-rat one-half)
1/2
> (define one-third (make-rat 1 3))
> (print-rat (mul-rat one-half one-third))
1/6
> (print-rat (add-rat one-third one-third))
6/9
```

課題 4-2: (既約分数)

上の定義のままだと、既約分数でないものができてしまう。そこで、(make-rat n d) を、n と d の最大公約数でそれぞれを割った商どうしで対を作るように make-rat の定義を改良せよ。ただし、n と d の最大公約数 (Greatest Common Divisor) は、関数 (gcd n d) によって求められるものとする。 ∎

課題 4-3: (正規化) さらに、(1) 負の分数を表現する場合に、必ず分子を負にして分母を正にする、(2) ゼロは 0/1 で表現する、のように改良せよ。改良した定義は例えばファイル rat-normal.scm に保存しておくと便利である。 ∎

1.3 抽象化による壁

これまで見てきたように、同じ「分数を扱う」と言っても様々な意識のレベルがある。データの抽象化では、「下位の内部表現を意識せずにデータを使うレベル」と「上位から隠されている内部表現のレベル」があり、それらの「壁」の間を構築子・選択子といった関数が橋渡しをしている関係がある。実際に大きなアプリケーションプログラムを書く場合には、こういったレベルが何段にも積み重ってできている。

このように、複数のレベルによってプログラムを構成することの利点は、

- 各レベルが独立している
- 複雑なデータ構造を考える際に、いきなり詳細まで考えずに済む

といったことである。

各レベルが独立していることの利点は、内部表現の定義を変えても、それを使う側の定義を変更する必要がないことである。例えば、「既約分数化」の課題で見たように分数の内部表現を変えても、使う側 (add-rat など) の定義を変更する必要がない。

課題 4-4: (既約分数化その 2)

先の課題では、分数を作る度に既約分数化していた。別の方法としては、make-rat で分数を作る際には、そのままの値を対にしておき、numer, denom で分母・分子をとり出す際に既約分数化する、というものが考えられる。

このアイデアに従って make-rat, numer, denom を定義し直せ。さらに、この定義を用いても add-rat の定義を変更する必要がないことを確認せよ。 ∎

課題 4-5: (さらに高レベルな使用)

上の課題で定義した分母による四則演算の関数を使って、ニュートン法によって平方根を求める関数 sqrt を、分数で計算するように定義を変更せよ。

ただし、四則演算のほかに、

- 分数の符号が正であることを調べる関数 (positive-rat? x)
- 分数 x, y が x < y であることを調べる関数 (less-than-rat? x y)
- 分数 x の絶対値を求める (abs-rat x)

などを必要に応じて定義せよ。また、いままでに作った分数の内部表現のどれを使っても、sqrt の結果が同じになるかどうかを調べてみよ。 ∎

1.4　データの意味

これまで、分数の内部表現として何種類かの定義を見てきた。これらはどれも妥当なようである。しかし、ある抽象データの内部表現を定義するとき、どうやって定義が妥当であると判断するのだろうか?

つまるところ、「使う側」がその抽象データとして期待しているような性質を満たしていればよいわけである。分数の場合であれば、

> (make-rat n d) の結果が x であるときに、
>
> $$\frac{\texttt{(numer x)}}{\texttt{(denom x)}} = \frac{\texttt{n}}{\texttt{d}}$$

という性質を満たしていればよい。

実際に、最初に示した定義がこの性質を満たしていることは、

$$\begin{aligned}
&\frac{\texttt{(numer x)}}{\texttt{(denom x)}} \text{ ただし } \texttt{x} = \texttt{(make-rat n d)} && (1)\\
&= \frac{\texttt{(numer x)}}{\texttt{(denom x)}} \text{ ただし } \texttt{x} = \texttt{(cons n d)} \quad (\texttt{make-rat} \text{ の定義より}) && (2)\\
&= \frac{\texttt{(numer (cons n d))}}{\texttt{(denom (cons n d))}} \quad (\texttt{x} \text{ を置き換え}) && (3)\\
&= \frac{\texttt{(car (cons n d))}}{\texttt{(cdr (cons n d))}} \quad (\texttt{numer}, \texttt{denom} \text{ の定義より}) && (4)\\
&= \frac{\texttt{n}}{\texttt{d}} \quad (\texttt{car}, \texttt{cdr}, \texttt{cons} \text{ の性質より}) && (5)
\end{aligned}$$

のようにして確められる。

また、対についても、

> 任意のデータ x, y について (cons x y) の結果が z であるときに、(car z) が x に、(cdr z) が y になる

という性質を満たせばよい。

Scheme において、対は原始的な関数によって提供されていると説明してきたが、必ずしもその必要はない。例えば、以下のような定義

```
(define (cons x y)
  (lambda (m)
    (cond ((= m 0) x)
          ((= m 1) y)
          (else (error "argument should be 0 or 1" m)))))
(define (car z) (z 0))
(define (cdr z) (z 1))
```

は、対の内部表現として「0 か 1 を受け取って car 部か cdr 部を返す関数」としている。これが対としての性質を満たしていることも容易に確かめられる。

課題 4-6: (対)

対の内部表現として、さらに違った方法もあり得る。

```
(define (cons x y)
  (lambda (f) (f x y)))
(define (car z)
  (z (lambda (a b) a)))
```

同様にして cdr を定義せよ。さらに、これが対としての性質を満たしていることを確かめよ。 ∎

2 階層データ

Cons は対を作る原始的な手段であり、どのようなデータでも対にすることができる。当然、対の対を作ってもよい。このことは、対を組み合わせてゆくことで、様々な種類のデータ構造を作れることを示唆している。

ここでは、代表的かつ基本的なデータ構造として、列と、構造の一部がその構造自身で作られているような階層的な (hierachical) データについて見てゆく。

2.1 列

列 (sequence) を作る場合、各要素について「car 部に要素が、 cdr 部に次の要素の対」が入っているようなリスト構造を作る。例えば、 1,2,3,4 という列は、

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

となる。(ただし、'() は「空のリスト」を作るための方法である。)

Scheme には、リストを作るための関数として list があり、上の例は

```
(list 1 2 3 4)
```

と書くこともできる。

リストに対する操作

リストに対する操作は、 cdr 部を手繰っていく形になるのが普通である。例えば、 n 番目の要素をとり出す list-ref は、

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

のようになる。

また、リストの長さを求める length は、

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

のようになる。ただし null? は、引数が「空のリスト」であるかどうかを調べる原始的な関数である。

2 つのリストをつなげる append は、

```
(define (append list1 list2)
  (if (null? list1)
      list2
      (cons (car list1) (append (cdr (list 1)) list2))))
```

のようになる。

課題 4-7: (last-pair)

リストの最後の要素だけを返す last-pair:

```
> (last-pair (list 23 72 149 34))
(34)
```

を定義せよ。(但し、引数として空でないリストを受け取るとする。) ∎

課題 4-8: (reverse)

リストを逆順にする関数 reverse:

```
> (reverse (list 1 4 9 16 25))
(25 16 9 4 1)
```

を定義せよ。 ∎

リストに対する写像

リストの各要素に対して何らかの変換をしたリストを作る、ということは頻出する操作である。

例えば、リストの各要素を定数倍する scale-list は:

```
(define (scale-list items factor)
  (if (null? items)
      '()
      (cons (* (car items) factor)
            (scale-list (cdr items) factor))))
```

のようになる。

すでに学んだように、「各要素を定数倍」の部分をパラメータ化することで、一般化した高階関数 map を定義することができて、

```
(define (map f items)
  (if (null? items)
      '()
      (cons (f (car items)) (map f (cdr items)))))
```

のようになる。 Scale-list は map を使って

```
(define (scale-list factor items)
  (map (lambda (x) (* x factor)) items))
```

のように書き直せる。

課題 4-9: (square-list)

数の列を受け取って、それぞれの要素を二乗した列を返す square-list を定義せよ。例えば、 (square-list (list 1 2 3 4)) は (1 4 9 16) となる。

```
;;; 定義 1
(define (square-list items)
  (if (null? items)
      '()
      (cons ⟨??⟩ ⟨??⟩)))

;;; 定義 2
(define (square-list items)
  (map ⟨??⟩ ⟨??⟩))
```

2.2 階層的構造

列の要素自身が列であるようなデータを扱うこともできる。例えば、 (cons (list 1 2) (list 3 4)) の値は、第 1 要素が (1 2)、第 2, 第 3 要素が 3,4 であるようなリストである。

このデータ構造は、木構造 (tree structure) と見ることもできる。木構造に対する計算は、再帰的に行うのが自然である。

例えば、木構造の葉 (末端の要素) の個数を数える関数 count-leaves は、

- 空のリストの count-leaves は 0

- ある木 x の count-leaves は、 x の car 部の count-leaves と cdr 部の count-leaves の和
- 葉の count-leaves は 1

のように考えることができる。

```
(define (count-leaves x)
  (cond ((null? x) 0)
        ((not (pair? x)) 1)
        (else (+ (count-leaves (car x))
                 (count-leaves (cdr x))))))
```

ただし pair? は、引数が cons によって作られたデータであるかどうかを調べる関数である。

課題 4-10: (deep-reverse)

関数 reverse に、各要素が列であるような列を与えても、内側の列の順序はそのままである。要素が列であった場合、それも逆順にするような deep-reverse を定義せよ。

```
> (define x (list (list 1 2) (list 3 4)))
> x
((1 2) (3 4))
> (reverse x)
((3 4) (1 2))
> (deep-reverse x)
((4 3) (2 1))
```

∎

課題 4-11: (fringe)

木構造を引数として受け取り、木の各要素が順番に並んだ列を返す関数 fringe を定義せよ。

```
> (define x (list (list 1 2) (list 3 4)))
> (fringe x)
(1 2 3 4)
> (fringe (list x x))
(1 2 3 4 1 2 3 4)
```

∎