ＡＰＬからＪへ－素数を求める問題を例として

西川　利男

ＡＰＬとＪ－この２つのプログラミング言語の創始者，Ken Iverson が今年10月に亡くなられた．ＡＰＬ／Ｊも次の世代へとページがめくられた，というべきであろうか．

一方，　日経ソフトウェア誌2005年１月号にプログラミング言語特集として，ＡＰＬとＪの紹介とともに素数を求めるプログラムが掲載されている，ということが先月の例会で横山暁，志村正人両氏によって報告された．

この問題は時宜を得た格好のテーマであり，またプログラム出典の著者[1]として，その解説をシンポジウム・チュートリアルに供したいと思う．

１．ＡＰＬとＪとの比較

ＡＰＬでは特殊なＡＰＬ文字を，Ｊでは通常の記号文字を用いているので，一見すると異なるように見える．しかし，いずれも Iverson の思想にもとづく配列処理の関数型言語であり，基本の考え方はまったく同じで変わることはない．ただ，用語などを違った名前で呼んでいるのでこれらを対比して示す．

ＡＰＬ	Ｊ
変数	名詞
関数	動詞
関数定義	明示的(explicit)定義
	暗黙的(tacit)定義
作用素	副詞
	接続詞

その他，ＪはＡＰＬの発展として出来たので，一般的にいってＡＰＬが簡明なのに比べ，Ｊは機能がやや複雑であるといえよう．

文献[1]　西川　利男「基礎からのＡＰＬ」サイエンスハウス(1991).

２．素数を求める－ＡＰＬのプログラム

２．１　配列テーブルによる方法　文献 [1], p. 101 ～ 104

```
      N←20
      (2=+/[1] 0= (⍳N)∘.|⍳N)/⍳N
 2  3  5  7  11  13  17  19
```

このようにＮ＝ 20とすれば20以下の素数が求められる．　⍳ Ｎにより１２…20の整数のベクトルが生成する．コード（∘ ・｜）は外積の作用素と剰余の関数であり，上のベクトルが２項関数として作用した結果は剰余のテーブルが得られる．次のコード0=では割り切れた (1)，割り切れない (0) のテスト・フラグテーブルが得られる．次にこのフラグテーブルのタテ方向 [1] の和 (+/) による低減として，テストの場合の数がベクトルとして求められる．この場合の数が，１とその数自身との２回起きるとき，これが素数である．上のコード(2=)では位置を示すフラグとして得られるから，これをかっこで囲んだ左引数とし，右引数には最初の１から２０のベクトルをコード (/) により取り出せば素数が得られる．

２．２　エラトステネスのふるいの方法　文献 [1], p. 141 ～ 142

```
      ∇ Z←PRIMETO N;S;M;NP
[1]   S←N*0.5
[2]   M←(N,1)⍴~N↑1
[3]   NP←1
[4]   L1:NP←NP+M[2;]⍳1
[5]   →(NP>S)/L2
[6]   M←((R←⌈N÷NP),NP)⍴M
[7]   M[1↓⍳R;NP]←0
[8]   →L1
[9]   L2:Z←(N⍴M)/⍳N
[10]  ∇
      PRIMETO 20
 2  3  5  7  11  13  17  19
```

エラトステネスのふるいの方法とは１からＮまでの数に対し，２から順次割って行き，割り切れたものを取り除く．これを続けて行くと，最後には素数だけが ’ふるい’ に残る，というのが原理である．

ここでは PRIMETO という名前の関数として定義している．繰り返しはＮの平方根 (N*0.5) まで行えば充分なのでこれを回数Ｓとする．[2] 行目ではＮ↑１により１を先頭にＮ－１個の０のベクトルが生成する．これを反転 (~) した値，

つまり先頭が０でＮ－１個の１のベクトルをＭとし，ふるいのフラグの初期値とする．このフラグの意味は０は合成数，１は素数を示す．最初の１は合成数ではないが，普通は素数とせず取り除く．ＮＰはテストする素数の候補を示す．[4] 行目では次の素数の候補をもってくる．[6] 行目ではフラグ配列のＭを素数ＮＰを列として reshape(ρ) により整形する．そして [7] 行目では配列Ｍの最右列の２番目からの項を０にする．つまり割り切れたとしてフラグ配列Ｍから取り除く．このあたりが ’ふるい’ 操作のポイントである．[4] 行目に戻ってフラグが１である次の素数候補を取り出し，次々と上の操作をくりかえして行く．なお．ＡＰＬでは繰り返し処理はループ構造ではなく，配列で一度に行うというのが原則だが，このように途中の計算結果により判定する場合ではループを使用することになる．[9] 行目で素数が取り出される．

３．素数を求めるーＪのプログラム

３．１　配列テーブルによる方法（ tacit プログラム）

```
        prime=. [:(#~2:=[:+/0:=|/~)>:&i.
        prime 20
2 3 5 7 11 13 17 19
```

アルゴリズムはＡＰＬの場合と全く同じである．ただこのＪプログラムでは引数をあらわに表示しない tacit 定義でなされているので，Ｊに慣れない人には戸惑いを感ずるだろう．

Ｊでは動詞，副詞，接続詞の組み合わせは，次のように機能する．

（動詞）（副詞）⇒（新しく出来た別の動詞）

（動詞）（接続詞）（動詞）⇒（新しく出来た別の動詞）

また，動詞が連続した場合，fork，hook というＪ独特の実行順序で行われる．

全体のプログラムは大きく３つの動詞から成っていて，この構造はforkと呼ばれる．

```
      [:  (#~2:=[:+/0:=|/~)  >:&i.
      __  ________________  _____
      f          g              h
```

そして引数Ｙを作用させたとき，つぎのように実行される．

(f Y) g (h Y)

さらに動詞ｆがcap([:) の場合，この動詞は何もしないという機能を持つので，

([: Y) g (h Y) ⇒ g(hY)

となり，結局hYの結果にｇを行うということになる． capped fork と呼ぶ．

まずｈの部分を見てみよう．複合した新しい動詞 >:&i. は i.20 で０～19の数ベクトルを >: により１～20 にする．ちなみにＪでは０オリジンでありこの操作が必要である．

次にかっこでくくられたg の部分を詳細に見てみる．Ｊでは右から演算されることを考慮し，内部の構造を分解してみるとかなり複雑で以下のようになる．

```
   #~ 2: = [: +/ 0: = |/~
                                  __ _ ___ (1)
                            __ __ _______ (2)
                      __ _ _____________ (3)
                __ ___________________ (4)
```

(1) の部分もforkで，先の h で得られた値をY'とすると実行は次のようになる．

```
   (0: Y') = (|/~Y')
```

ここで副詞 (~) は１項動詞を同じ引数で２項動詞として働かせる．したがって，右の複合動詞 (|/~) の部分は実際は

```
   1 2 … 20 |/ 1 2 … 20
```

と剰余のテーブルが作られる．左の動詞(0:)は０を生成する動詞であり，結局テーブルの値が０になるかどうかのテストテーブルが得られる．(2) の部分も capped fork でこのテストテーブルの和(+/)を計算し結果はベクトルになる．さらに (3) の部分もforkであり２に等しいかのフラグを得る．(4) の部分は動詞が２つ連なったものでこれはhookと呼ばれ次の働きをする．

```
   (g h) Y'' ⇒ Y'' g (h Y'')
```

副詞 (~) は２項動詞に作用すると，左右の引数を交換する働きをもつ．副詞が作用する動詞が１項か２項かによって機能が異なることに注意．結局，

```
   ( (3) で得た 2 に等しい位置のベクトル)  #  ( 1 ～20のベクトル)
```

となり，素数が取り出されたことになる．このようにＪのコーディングのアルゴリズムも先のＡＰＬのものと全く一致している．

３．２　エラトステネスのふるいの方法（explicitプログラム）

```
sieve=: 3 : 0
r=. y.^0.5
s=.-.y.{.1
wr (>:i.y.),: s
p=.0
 while. r>:p=. p+>:(p}.s)i.1 do.
 s=.0 (<:}.p*>:i.<.y.%p)}s
wr p
 wr >:q=.<:}.p*>:i.<.y.%p
wr (>:i.y.),: s
end.
wr 'primes:'
>:s#i.y.
)
```

```
wr=: 1!:2&2    NB. print
```

同じＪのプログラムでもexplicit定義では引数をy.としてあらわに示し，また途中の処理の値も示されるので，初めての人にもずっと容易で，かつ安全なプログラムが出来る．

第１行では sieve という名前で動詞 (3) として定義を開始し，最後の閉じかっこ）までがその内容である．なお，名詞 (0)，副詞 (1)，接続詞 (2) も定義できる．

各行のコーディングはＡＰＬと対応しているので，それぞれの記号文字を比較対照して見れば理解は容易であろう．

Ｊではループ構造として

```
while.（条件テスト）do.（ 実行処理）end.
```

さらには条件分岐などいろいろな構造が備わっている．

ひとつだけ注意すべきコーディングを指摘しておく．

```
s=.0 (<:}.p*>:i.<.y.%p)}s
```

これはＪでは amend （修正）と呼ばれる副詞(}) を用いるもので

（変更値）（変更位置インデックス）}（元の配列）

として配列の中の位置を指定して値を変更する．ただし，変更した値が自動的に配列になるのではなく，あらわに代入することによって配列を修正する．これは関数型言語として，副作用をさけるための配慮であるが，使用法に注意を要する．

最後の行は素数の位置フラグから素数を取り出すものだが，位置フラグなどインデックスにＪでは０－オリジンであるため１だけインクリメント(>:)が必要である．

以下，実行のようすを示すが，’ふるい’の途中経過も表示するようにして，プログラムの動きが分かるようにした．

```
      sieve 20
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
 0  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
2
4 6 8 10 12 14 16 18 20
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
 0  1  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0
3
6 9 12 15 18
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
 0  1  1  0  1  0  1  0  0  0  1  0  1  0  0  0  1  0  1  0
 primes:
2 3 5 7 11 13 17 19
```

３．３　プリミティブp:による方法

```
        primes_to =: i.&.(p:^:_1)
        primes_to 20
2 3 5 7 11 13 17 19
```

Ｊではシステムに　p:　として，素数を求める（原始動詞）が備えられている．しかし，p: 20 のように用いたときには，20番目（0-オリジンで）の素数が得られるだけである．上のようにすると 20　までの素数が求められる．このコーディングの意味を考えてみよう．

これは大きく次の構造である．

```
    i.      &.      (p:^:_1)
    ──   ─────   ──────
   動詞     接続詞          動詞
```

ここで接続詞(&.)は under と呼ばれ左右の引数動詞u，vに対し次の働きをする．

$(u\ \&.\ v)Y \Rightarrow v^{-1}\ (u\ (v\ Y))$

ここで v^{-1} は　v の逆演算を示す．

さて，右の動詞部分を見てみる．

```
    p:  ^:  _1
        ──  ──  ──
```

接続詞(^:)は power と呼ばれ，右引数として　_1　をとったときは，左引数の動詞の逆演算を行う．ここでは20を越えない最大の素数が何番目になるかのインデックスが与えられる．つまり

$(p:^:\_1)\ 20 \Rightarrow 8$

したがって，最後の動作は次のようになるので，素数が求められる．

```
    p: i. 8
2 3 5 7 11 13 17 19
```

４．最後にひと言

ＡＰＬもＪもつまるところ考え方は同じある．いわんや，Ｊのtacit 定義とexplicit定義とどちらが良いかなどは，個人の好みの問題である．

幸いわれわれ日本人は漢字，ひらがな，カタカナさらにABC と世界にもまれなさまざまな文字を駆使して文章を作ってきた．また，文体にしても詩歌の韻文もあれば日常の散文もある．ＡＰＬ，Ｊのそれぞれがどれに似ているかは正月の座興としよう．問題はプログラムの中味であり，表現方法がいろいろ選択できるというのもまた楽しみではないか，私は思っている．