プログラミング言語論第11回
遅延評価とストリーム

情報工学科 山本修身

遅延評価について

遅延評価 (lazy evaluation) とは，評価することが必要になるまで式の評価を遅らせるためのしくみである．このしくみを用いて結果的には評価する必要がなかった式を評価しないで済ますことができる．

たとえば，適当な関数fを用いて定義されているベクトル ((f 1) (f 2) (f 3)) があったとする．このベクトルの第１成分と第３成分の和が結果的に必要であったということならば，(f 2) を計算する必要は結果的になかったことになる．予め評価を完了させるのではなくて，評価をしたように見せかけて，実際に評価することが必要になるまで評価しないのが遅延評価である．

```
(list (f 1) (f 2) (f 3))   ←すべての計算を予めやってしまう

(list (delay (f 1)) (delay (f 2)) (delay (f 3)))
                          ↑すべての計算を予めやったように見せかける．実際にはやっていない．あとで必要になったらやる予定．
```

遅延評価のためのしくみ

遅延評価のための関数がいくつか用意されている．遅延評価は評価したような振りをして実際には評価せず，必要になったとき評価するもので，評価した振りをしたデータを**プロミス（promise: 約束）**と呼ぶ．ある式からその式を評価するプロミスを作る特殊形式として，**delay** が与えられている．また，プロミスから値を求める関数として **force** がある．

```
(define a (delay
            (let ((a 2)
                  (b 3))
              (display "now being evaluated...")
              (newline)
              (+ a b))))
(define c a)
(display "the value has been assigned to c.")
(newline)
(define b (force c))
(display "value = ")
(display b)
(newline)
```

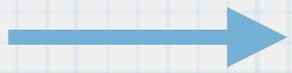
```
OMacBook:yama507> kawa test-delay.scm
the value has been assigned to c.
now being evaluated...
value = 5
OMacBook:yama508>
```

遅延評価をlambdaで実現する

遅延評価はlambdaを用いて実現することが可能である．遅延させるには，lambdaの衣で式を包む（このとき，評価をしてはいけない）．そのためにマクロを用いる（マクロについては．第13回の講義で説明する）．以下にmydelayとmyforceのプログラムを示す．これ以降示す例はこの実現方法で問題ないが，普通はSchemeにネイティブな実装が用意されているので，それを用いる．

```
(define-macro (mydelay expr) `(lambda () ,expr))
(define (myforce promise) (promise))

(define foo (mydelay
              (begin
                (display "being evaluated...")
                (newline)
                (+ 2 3))))
(display foo)
(newline)
(display (myforce foo))
(newline)
```



```
#<procedure foo>
being evaluated...
5
```

素数の和を求める (1)

ある整数aからb - 1までの区間に含まれる素数の和を計算することを考える．まず，素数であるか否かは以下のような関数 prime? で判定することができる．

```
(define (prime? n)
  (define (prime?-iter i)
    (cond ((> (* i i) n) #t)
          ((zero? (remainder n i)) #f)
          (else (prime?-iter (+ i 1)))))
  (prime?-iter 2))
```

ある数が素数であるということは，1とその数自身以外の整数で割り切れない数であるが，i = 2から順にその数 n を割っていって，i2 > nとなるところまで確認すれば良い．もしなんらかの数で割り切れるのであれば，それまでに適当な i で割り切れるはずである．

素数の和を求める (2)

prime?を用いて，a から b - 1 の素数の和を求めるには，以下のようなプログラムを書けば良い.

```
(define (sum-primes a b)
  (define (sum-primes-iter i res)
    (if (>= i b) res
        (sum-primes-iter
         (+ i 1)
         (if (prime? i) (+ i res) res))))
  (sum-primes-iter a 0))
```

名前付きletで書けばもう少し単純化される.

```
(define (sum-primes2 a b)
  (let sum-primes-iter ((i a) (res 0))
    (if (>= i b) res
        (sum-primes-iter
         (+ i 1)
         (if (prime? i) (+ i res) res)))))
```

```
(display (sum-primes2 100 1000))
(newline)
```

```
OMacBook:yama577> kawa prime.scm
75067
```

素数の和を求める (3)

ここまでのプログラミングに問題はないが，以前説明した高階関数を用いて，このプログラミングをさらに単純化してみる.

まず，aからスタートしてb未満の整数のリストを生成する関数 enumerate を以下のように定義する.

```
(define (enumerate a b)
  (if (= a b) '()
      (cons a (enumerate (+ a 1) b))))
```

```
(display (enumerate 10 100))
(newline)
```

```
OMacBook:yama502> kawa list_comp.scm
(10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35
 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61
 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87
 88 89 90 91 92 93 94 95 96 97 98 99)
```

素数の和を求める (4)

関数 enumerate を用いて，a以上b未満の素数の和を求める関数は以下のように定義することができる.

```
(define (sum-primes3 a b)
  (foldl + 0
         (filter prime? (enumerate a b))))
```

この関数は「a以上b未満の整数のリストを作って，それのうち素数のものだけフィルターしてそれをすべて足し合わせる. このようなプログラムは見やすいし，考えやすい.

```
(display (sum-primes3 100 1000))
(newline)
```

```
75067
```

遅延評価によってストリームを作る (1)

素数の和を計算するとき，`(sum-primes3 100 100000)`のように大きな区間を指定すると，その区間の幅の分だけ長いリストを生成する．これは効率が悪い．実際に足される数はリストの要素の一部である．そこで，一度にリストを作るのではなくて，実際にはリストを構成せずに，`delay`を用いて作ったふりをする．

まず，`enumerate`のプログラムを以下のように変える．

```
(define (enumerate a b)
  (if (= a b) '()
      (cons a (delay (enumerate (+ a 1) b)))))
```

delayを入れることによって，すぐには評価されないので，すぐに答が返ってくる．このようにcdr部が遅延されているようなリストをここでは **（遅延）ストリーム (lazy stream)** と呼ぶことにする．

遅延評価によってストリームを作る (2)

ここで作られたデータの先頭のn個をリストにするための関数takeを以下のように定義する.

```
(define (take n lst)
  (if (zero? n) '()
      (cons (car lst) (take (- n 1) (force (cdr lst))))))
```

以下の例では100万までの整数のリストを作っているが，遅延が入っているので，実際には計算されない．関数takeが要素を取り出すときに実際の計算が実行される.

```
(display (take 10 (enumerate 10 1000000)))
(newline)
```

```
(10 11 12 13 14 15 16 17 18 19)
```

遅延評価によってストリームを作る (2)

さらに，関数 filter もストリーム対応にさせてみる.

```
(define (stream-filter pred lst)
  (if (null? lst) '()
      (let ((cc (car lst))
            (dd (delay (stream-filter pred (force (cdr lst))))))
        (if (pred cc) (cons cc dd) (force dd)))))
```

```
(display (take 10 (stream-filter prime? (enumerate 10 1000000))))
(newline)
```

```
(11 13 17 19 23 29 31 37 41 43)
```

遅延評価によってストリームを作る (3)

素数の和を計算するために，ストリームから和を計算する関数stream-foldlを定義する.

```
(define (stream-foldl op g lst)
  (if (null? lst) g
      (stream-foldl op (op g (car lst)) (force (cdr lst)))))
```

この関数を用いて，100から1000000までの素数の和を計算すると以下のようになる.

```
(display
 (stream-foldl + 0
               (stream-filter prime? (enumerate 10 1000000))))
(newline)
```

15秒ほどで答えが得られた最初に示したsum-primesとほとんど変わらない時間で計算できている.

```
37550400963
```

遅延評価によってストリームを作る (4)

同じ計算をリストでやると以下のように，スタックオーバフローのエラーが発生して止まってしまう.

```
OMacBook:yama520> kawa list_comp.scm
Exception in thread "main" java.lang.StackOverflowError
        at gnu.math.IntNum.add(IntNum.java:431)
        at gnu.kawa.functions.AddOp.apply2(AddOp.java:57)
        at gnu.kawa.functions.AddOp.applyN(AddOp.java:145)
        at gnu.kawa.functions.AddOp.applyN(AddOp.java:160)
        at gnu.mapping.ProcedureN.apply2(ProcedureN.java:39)
    at list_comp.enumerate$X(list_comp.scm:22)
    at list_comp.enumerate$X(list_comp.scm:22)
    at list_comp.enumerate$X(list_comp.scm:22)
.........
```

関数 enumerate の内部ですでにメモリー不足が起こってしまう. 長さ100万のリストに要するメモリ（特にこれはスタック上のメモリである）は莫大なものとなる.

無限に長いストリームで計算する (1)

計算機のメモリーは有限であるので，無限の長さを持つリストを直接メモリー上で表現することは不可能である．しかし，論理的に無限の長さを持つストリームをdelayを用いて作ることは可能である．まず，自然数のストリームを作る．

```
(define (make-integer)
  (let loop ((i 0))
    (cons i (delay (loop (+ i 1))))))
(define integer (make-integer))
```

変数 integer は0から無限に伸びているリストを表している．実際，最初の20個の値をとってみると以下のようになる．

```
(display (take 20 integer))
(newline)
```

```
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19)
```

無限に長いストリームで計算する (2)

このストリームを用いて，素数全体のストリームを作ることができる．これは以下のように定義できる．

```
(define (stream-cdr lst) (force (cdr lst)))

(define primes (stream-filter prime?
                              (stream-cdr
                               (stream-cdr integer))))
```

stream-cdrを2回かけているのは，0と1を除外して2以上の整数の列を使うためである．ストリームのn番目の要素を取り出す関数steam-nthを以下のように定義すれば，10000番目の素数をそのまま計算することができる．

```
(define (stream-nth n lst)
  (if (zero? n) (car lst)
      (stream-nth (- n 1) (force (cdr lst)))))

(display (stream-nth 10000 primes))
(newline)
```

→ 104743

素数のストリームのエラトステネスの篩としての定義

関数prime?はいちいち割り算をしているが，素数の性質を十分に利用しているとは言えない．そこで，エラトステネスのふるいのアルゴリズムでストリームを構成することを考える．これが効率的かどうかは疑問であるが，ストリームを構成することは可能である．

```
(define (make-primes lst)
  (let ((m (car lst)))
    (cons m
          (delay
            (make-primes
             (stream-filter (lambda (n) (not (zero? (remainder n m))))
                            (force (cdr lst))))))))
(define primes2 (make-primes (stream-cdr (stream-cdr integer))))
```

```
(display (take 20 primes2))
(newline)
```

```
(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71)
```

フィボナッチ数列を定義する (1)

フィボナッチ数列は以下のように定義される数列のことである.

$$F_{n+2} = F_{n+1} + F_n,\ F_0 = F_1 = 1$$

この数列の中身を表すストリームを考えてみる．まず，２つのストリームの対応する中身を足し合わせる関数を以下のように定義する.

```
(define (stream-add lst1 lst2)
  (cons (+ (car lst1) (car lst2))
        (delay (stream-add (force (cdr lst1))
                           (force (cdr lst2))))))
```

フィボナッチ数列を定義する (2)

フィボナッチ数列は以下のような性質がある.

$$
\begin{array}{lccccccc}
 & F_0 & F_1 & F_2 & F_3 & F_4 & F_5 & \cdots \\
+) & & F_0 & F_1 & F_2 & F_3 & F_4 & \cdots \\
\hline
 & F_2 & F_3 & F_4 & F_5 & F_6 & \cdots
\end{array}
$$

この性質より，フィボナッチ数列のストリームは以下のように定義することができる.

```
(define (make-fib)
  (cons 1 (delay
            (cons 1 (delay
                      (stream-add fib-stream
                                  (force (cdr fib-stream))))))))
(define fib-stream (make-fib))
```

実際 fib-streamの内容を見てみると以下のようになる.

```
(display (take 40 fib-stream))
(newline)
```

```
(1 1 2 3 5 8 13 21 34 55 89 144 233 377
610 987 1597 2584 4181 6765 10946 17711
 28657 46368 75025 121393 196418 317811
514229 832040 1346269 2178309 3524578
 5702887 9227465 14930352 24157817
39088169 63245986 102334155)
```

フィボナッチ数列を定義する (3)

もちろん，前のスライドような性質を使わなくてもフィボナッチ数列は定義の漸化式から以下のように定義することもできる.

```
(define (genfib a b)
  (cons a (delay
            (genfib b (+ a b)))))
(define fib-stream2 (genfib 1 1))
```

a，bを初期値とするフィボナッチ数列のストリームを作る

このストリームの中身を調べるとフィボナッチ数列になっている.

```
(display (take 20 fib-stream2))
(newline)
```

```
(1 1 2 3 5 8 13 21 34 55 89 144 233
377 610 987 1597 2584 4181 6765)
```

$$F_{n+2} = F_{n+1} + F_n,\ F_0 = F_1 = 1$$

乱数を作り出すストリームとモンテカルロ法 (1)

Schemeには乱数を作り出す関数が標準では用意されていないので，ここではJavaの関数を用いる．以下のようにして，乱数を作り出すストリームを定義することができる．

```
(define (random) (java.lang.Math:random))

(define (make-random-stream)
  (cons (random)
        (delay
          (make-random-stream))))
(define random-stream (make-random-stream))
```

```
(display (take 10 random-stream))
(newline)
```

```
(0.417382451526255 5 0.450716826628402 5 0.844288535192817 7 0.578039768863541 4
 0.00451671753188087 6 0.394654473604584 9 0.601609508877820 7 0.693480634666055 4
 0.542979160608899 2 0.64504140070802 7)
```

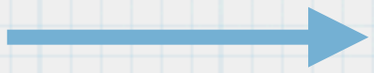
乱数を作り出すストリームとモンテカルロ法 (2)

乱数列を使ってモンテカルロ法で円周率を計算してみる．まず，乱数列から2つずつ数をとってきて座標を並べたストリームを作る．

```
(define (make-pair-stream s)
  (let ((s0 (car s))
        (s1 (car (force (cdr s)))))
    (cons (cons s0 s1)
          (delay
            (make-pair-stream
              (force (cdr (force (cdr s)))))))))

(define random-pair-stream (make-pair-stream random-stream))
```

```
(display (take 10 random-pair-stream))
(newline)
```

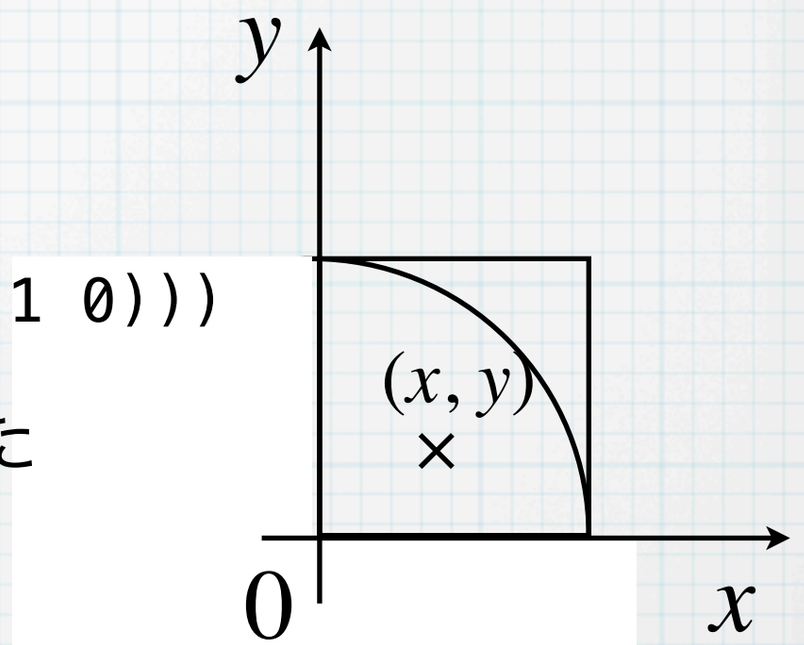


```
((0.0758897014982918 8 . 0.253570193405558 2)
 (0.463567016558224 2 . 0.0975528937941305 7)
 (0.965520601192959 5 . 0.592125233921287 9)
 (0.256096961967337 8 . 0.310651613911499)
 (0.856159081802443 8 . 0.764004188468802)
 (0.644767183733942 8 . 0.973208542716302 1)
 (0.012074698828168362 . 0.4740827876732127 7)
 (0.619603039658066 7 . 0.0805942361365872 3)
 (0.985589018014915 8 . 0.0602898542270917 66)
 (0.591251598496303 8 . 0.653512146143164 8))
```

乱数を作り出すストリームとモンテカルロ法 (3)

点の座標が与えられたとき，その点が半径１の4分の1円の中に入っているか否かを判定する関数 in-the-circle? を定義する.

```
(define (in-the-circle? pair)
  (let ((x (car pair))
        (y (cdr pair)))
    (if (< (+ (* x x) (* y y)) 1.0) 1 0)))
```



この関数をストリームのすべての要素に適用した値を要素とするストリームを作る. このために stream-map を定義する.

```
(define (stream-map func lst)
  (cons (func (car lst))
        (delay
          (stream-map func (force (cdr lst))))))

(define circle-stream (stream-map in-the-circle? random-pair-stream))
```

```
(display (take 30 circle-stream))
(newline)
```

```
(1 1 1 1 0 1 1 1 1 0 1 1 1 1 1 1
1 0 1 1 1 1 1 0 1 0 1 1 1 1)
```

乱数を作り出すストリームとモンテカルロ法 (4)

前のスライドで定義した circle-stream は0, 1の列が出てくるストリームである．1の出現した割合を計算するために，i番目の要素について (Σ v) / (i + 1)を計算するストリームを考える． ただし，Σ v は i 番目までに出現した1の個数である．このようなストリームを計算するために，以下のような関数を定義する．

```
(define (make-ratio-stream sum num_of_data lst)
  (let ((sum1 (+ sum (car lst)))
        (num_of_data1 (+ num_of_data 1)))
    (cons (/ sum1 num_of_data1)
        (delay
          (make-ratio-stream sum1 num_of_data1 (cdr lst))))))

(define ratio-stream (make-ratio-stream 0.0 0.0 circle-stream))
```

以下のように200000番目の要素を取り出して4倍すると円周率の近似値になっている.

```
(display (* 4 (stream-nth 200000 ratio-stream)))
```

→ `3.144124279378603`

並列処理のためのしくみ (1)

Schemeにおいては，遅延に似たしくみとして future という特殊形式が用意されている．関数delayは与えられた式をすぐに評価せずにプロミスを返したが，futureは与えられた式の別のスレッドでの実行を開始し，値としてスレッドを返す．別スレッドで実行された式の評価の結果を得るためには，返って来たスレッドをforceすれば良い．ただし，式の評価が終了していない場合には，実行がブロックされる（待たされる）ことになる．

(future 式) => バックグラウンドで動くスレッド

(force スレッド) => スレッドの返した値

並列処理のためのしくみ (2)

関数の実行時間を計測する関数 time を用いて，フィボナッチ数列の計算に要する時間を2種類のプログラムで計算してみる．一つは定義通り計算し，もう一つは，(fib 36) と(fib 37)を並列に計算して，最後に足し合わせるものである．実計算時間が明らかに変化する．

```
(define (fib n)
  (if (<= n 1) 1
      (+ (fib (- n 1)) (fib (- n 2)))))
(display (time (fib 38)))
(newline)
(display
 (time
  (let ((v1 (future (fib 37))))
    (let ((v2 (fib 36)))
      (+ (force v1) v2)))))
(newline)
```

このプログラムは第13回の講義で説明するのでわからなくても良い→

```
kawa fib3.scm
Time = 41216 ms
63245986
Time = 25674 ms
63245986
```

実験に使用した計算機のCPUはdual core

```
(define-macro (time exp)
  `(let ((t1 (java.lang.System:currentTimeMillis)))
     (let ((val ,exp))
       (let ((t2 (java.lang.System:currentTimeMillis)))
         (display "Time = ")
         (display (- t2 t1))
         (display " ms")
         (newline)
         val))))
```