

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

遅延評価をlambdaで実現する

4

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
```

遅延評価について

2

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せることができる。

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。
```

素数の和を求める (1)

5

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を割って行って、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。

遅延評価のためのしくみ

3

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>
```

素数の和を求める (2)

6

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)

7

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)
```

遅延評価によってストリームを作る (2)

10

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、遅延が入っているため、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)
```

素数の和を求める (4)

8

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、そのうち素数のものだけフィルタ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 (2)

11

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 (1)

9

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、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ため、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 (3)

12

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 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なになる。

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

```
37550400963
```

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。

遅延評価によってストリームを作る (4)

13

同じ計算をリストでやると以下のように、スタックオーバー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$(list_comp.scm:22)
  at list_comp.enumerate$(list_comp.scm:22)
  at list_comp.enumerate$(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ー（特にこれはスタック上のメモリーである）は莫大なものとなる。

素数のストリームのエラトステネスの篩としての定義

16

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)
```

無限に長いストリームで計算する (1)

14

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)
```

フィボナッチ数列を定義する (1)

17

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))))))

```

無限に長いストリームで計算する (2)

15

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 `stream-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
```

フィボナッチ数列を定義する (2)

18

フィボナッチ数列は以下のような性質がある。

	F_0	F_1	F_2	F_3	F_4	F_5	...
+	F_0	F_1	F_2	F_3	F_4	...	
	F_2	F_3	F_4	F_5	F_6	...	

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)
```


並列処理のためのしくみ (2)

25

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)
```

→

```
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))))))
```

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→