プログラミング言語論第12回
例題による解説 (2)

情報工学科 山本修身

SchemeをSchemeで書いてみる

言語処理系を作ることは容易なことではない．しかし，Scheme処理系をSchemeで記述するのはそれほど難しくない．多少コード量が増えるが1ステップずつ見ていくことにする．一般に現在開発している言語をその言語自身で記述するということはよく行われることで，実際にうまくいっている有名な例としては pypy がある．

pypyは本来のpythonよりも高機能な処理系となっている．高級言語で高級言語を記述すると，比較的少ないコード量で記述することが可能となる．

pypy.org (pypyのホームページ)

Scheme → Scheme

これから作るもの

Scheme → Java

kawaの構造

Python → Python

pypyの構造

Schemeの外枠 REPLを仮に書いてみる

REPL (**r**ead-**e**val-**p**rint **l**oop) はコンソールからS式を読んでそれを評価して，表示するループのことで，Schemeの一番外側の（ユーザに一番近い）部分を構成する．まず，evalの部分を除いてREPLを書いてみる．以下のREPLは「オウム返し」するだめのものである．

```
(define (repl)
  (display "my-scheme> ")
  (let ((exp (read)))
    (when (not (eq? exp #!eof))
      (display "==> ")
      (display exp)
      (newline)
      (repl))))
(display "** Scheme on Scheme **")
(newline)
(repl)
(display "Bye, see you.")
(newline)
```

←ここではそのままexpを表示しているが，最終的には(eval exp env)として式を評価する．

関数 read はコンソールからS式を一つ読み込んで返す関数である．kawaのreadは読み込む前に改行を出力してしまうので右のようになっている．

```
OMacBook:yama511> kawa scheme.scm
** Scheme on Scheme **
my-scheme>
(+ 1 2 3)
==> (+ 1 2 3)
my-scheme>
'(a b c)
==> (quote (a b c))
my-scheme>
Bye, see you.
OMacBook:yama512>
```

式を評価するための仕組み：eval (1)

式を評価するためには，与えられた式を分解して，それぞれの種類に応じて，式→式の変換を行えば良い．ここでは入力も出力もS式のみを考えている．ここで考える式は以下のものに限定する（実際のScheme処理系ではもっといっぱい定義される）：

- 数や文字列，ブール値などのデータはそのまま返す
- シンボルは変数であると考え，それに対応する値を環境から探し出してその値を返す
- クオートはそのまま中身を返す
- `set!`は環境から束縛を探し出して値を変更する
- `define`は現在のフレームに束縛を作る．
- `if`は条件に応じて式を評価する処理を行う．
- ラムダ式はプロセジャー作る．
- `begin`は順に中身を評価して最後の値を返す．

プログラムリスト9〜31行目を参照すること

式を評価するための仕組み：eval (2)

- `cond`は`if`に書き換えてから評価する（syntactic sugar).
- `let`は`lambda`式に書き換えてから評価する（syntactic sugar).
- 関数への適用については`apply`を呼び出す.
- それ以外の構造についてはエラーを出力する.

このプログラムはScheme処理系の作り方を説明するためのものなので，最低限の機能を実現している．本格的なScheme処理系にするにはまだいくつか基本的な文法などを付加する必要がある.

プログラムリスト9〜31行目を参照すること

式を評価するための仕組み：eval (3)

実際のプログラムは以下のようになる.

11 行目

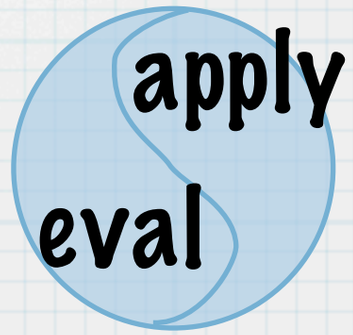
```
(define (eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp)
         (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp)
                         (lambda-body exp)
                         env))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env))
        ((cond? exp) (eval (cond->if exp) env))
        ((let? exp) (eval (let->lambda exp) env))
        ((application? exp)
         (apply (eval (operator exp) env)
                (list-of-values (operands exp) env)))
        (else
         (error "Unknown expression type -- EVAL" exp))))
```

関数適用のしくみ (1)

exalが式を評価する場合，その式が関数への適用であればapplyを呼び出す．applyに引数を適用する場合，組み込み関数であればそのままデータを関数に渡して関数適用する．lambdaによって表現されている場合，第10回で説明したように，新たなフレームを作ってから，そのlambda式が評価された環境でlambdaの本体を評価する．

34 行目

```
(define (apply procedure arguments)
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           arguments
           (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY" procedure))))
```



関数適用のしくみ (2)

組み込み関数は，lambdaによって定義された手続きではなく，Schemeシステムを記述しているレベルで定義されたもので，直接呼び出されるものである．大域環境上で定義されるときは，(primitive ベースになるSchemeシステムの関数）という形で格納される．

```
(define apply-in-underlying-scheme apply)
```

7 行目

myscheme内部でapplyを定義しているので本当のapplyを上記のように見えるようにする．

```
  (define primitive-procedures
    `((car ,car) (cdr ,cdr) (cons ,cons) (null? ,null?)
      (* ,*) (+ ,+) (- ,-) (/ ,/) (= ,=) (> ,>) (>= ,>=) (< ,<)(<= ,<=)
      (not ,not) (and ,and) (or ,or) (display ,user-print)
(newline ,newline)
      (true? ,true?) (false? ,false?)))
  (define (primitive-procedure-names) (map car primitive-procedures))
  (define (primitive-procedure-objects)
    (map (lambda (proc) (list 'primitive (cadr proc)))
         primitive-procedures))
  (define (apply-primitive-procedure proc args)
    (apply-in-underlying-scheme
     (primitive-implementation proc) args))
```

294 行目

関数適用のしくみ (3)

lambdaによって定義された手続き

```
((compound-procedure? procedure)
 (eval-sequence
  (procedure-body procedure)
  (extend-environment
   (procedure-parameters procedure)
   arguments
   (procedure-environment procedure))))
```

37 行目

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps)(eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

61 行目

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))
```

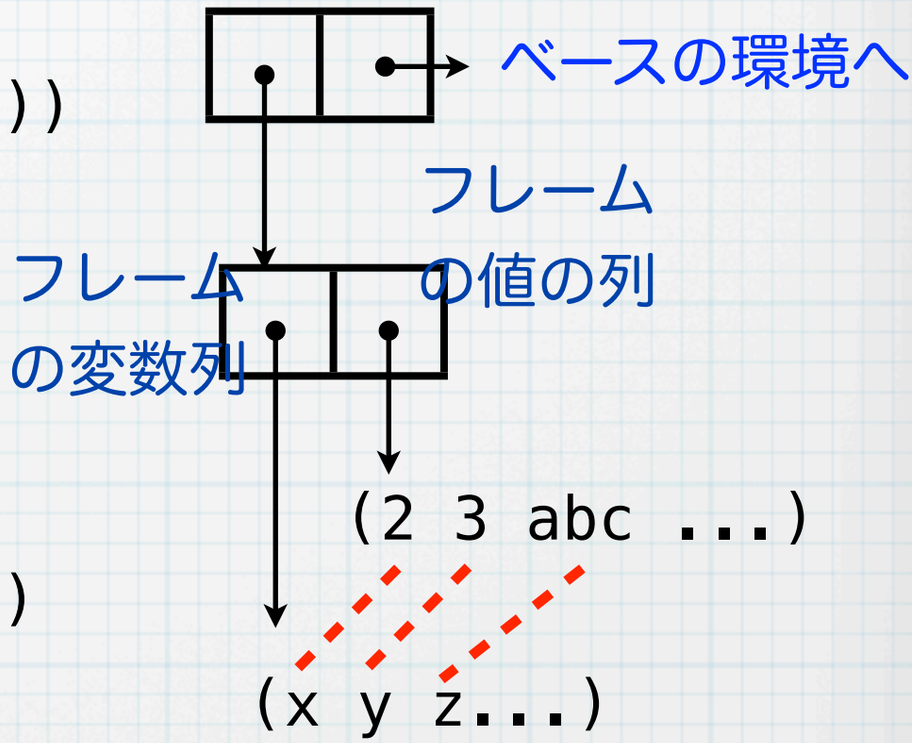
237 行目

環境の実装 (1)

式を評価する場合，関数を起動する場合の基礎になるのが数回前の講義で説明した環境である．ここではまず環境のしくみを作る．

227 行目

```
(define (enclosing-environment env) (cdr env))
(define (first-frame env) (car env))
(define the-empty-environment '())
(define (make-frame variables values)
  (cons variables values))
(define (frame-variables frame) (car frame))
(define (frame-values frame) (cdr frame))
(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals))))
```



環境の実装 (2)

環境から変数を探し出すには以下のようにする.

259 行目

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (car vals))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbounded variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

束縛の値を返す

変数の処理

13 行目

```
((variable? exp)
 (lookup-variable-value exp env))
```

クオートの処理

15 行目

```
((quoted? exp) (text-of-quotation exp))
```

92 行目

```
(define (quoted? exp) (tagged-list? exp 'quote))
(define (text-of-quotation exp) (cadr exp))
```

環境の実装 (3)

set!のための環境の操作は以下のようになる.

243 行目

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
             (env-loop (enclosing-environment env)))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable -- SET!" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
                (frame-values frame)))))
  (env-loop env))
```

束縛の内容を変更する

set!の処理

67 行目

```
(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                       (eval (assignment-value exp) env)
                       env)
  'ok)
```

環境の実装 (4)

defineのための操作は以下のとおり.

274 行目

```
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond ((null? vars)
             (add-binding-to-frame! var val frame))
            ((eq? var (car vars))
             (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame)
          (frame-values frame))))
```

変数が見つからなければ
バインディングを追加

変数が見つかれば，その
値を変更する

defineの処理

74 行目

```
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
    (eval (definition-value exp) env)
    env)
  (list 'ok ': (definition-variable exp) 'is 'defined))
```

数や文字列などの処理

数や文字列などを評価しても，変化しない．そのようなデータはそのまま出力する．

12 行目

```
(cond ((self-evaluating? exp) exp)
```

81 行目

```
(define (self-evaluating? exp)
  (cond ((number? exp) #t)
        ((string? exp) #t)
        ((eq? exp #t) #t)
        ((eq? exp #f) #t)
        (else #f)))
```

```
** Scheme in Scheme **
my-scheme>
23.5
==> 23.5
my-scheme>
7
==> 7
my-scheme>
"meijo"
==> meijo
my-scheme>
```

beginの処理

beginでは式を順番に評価していく

23 行目

```
((begin? exp)
 (eval-sequence (begin-actions exp) env))
```

149 行目

```
(define (begin? exp) (tagged-list? exp 'begin))
(define (begin-actions exp) (cdr exp))
(define (last-exp? seq) (null? (cdr seq)))
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))
(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))
(define (make-begin seq) (cons 'begin seq))
```

この関数は後で何回か利用する→

61 行目

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps)(eval (first-exp exps) env))
        (else (eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

ifとcondの処理

ifの処理

18 行目

```
((if? exp) (eval-if exp env))
```

55 行目

```
(define (eval-if exp env)
  (if (true? (eval (if-predicate exp) env))
      (eval (if-consequent exp) env)
      (eval (if-alternative exp) env)))
```

condの処理

24 行目

```
((cond? exp) (eval (cond->if exp) env))
```

179 行目

```
(define (cond->if exp)
  (expand-clauses (cond-clauses exp)))
(define (expand-clauses clauses)
  (if (null? clauses)
      'false
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF"
                       clauses))
            (make-if (cond-predicate first)
                     (sequence->exp (cond-actions first))
                     (expand-clauses rest))))))
```

condは一度ifに書き換えてから評価する.

condは単純な書き換えなので，出現のたびに書き換えることになる．このような式を糖衣構文（syntactic sugar)と呼ぶ.

lambdaの処理

lambdaを評価するとプロセジャーが生成される．プロセジャーはパラメータ列，本体，環境の組としてリストを生成する．

lambdaの処理

19 行目

```
((lambda? exp)
 (make-procedure (lambda-parameters exp)
                 (lambda-body exp)
                 env))
```

214 行目

```
(define (make-procedure parameters body env)
  (list 'procedure parameters body env))
(define (compound-procedure? p)
  (tagged-list? p 'procedure))
(define (procedure-parameters p) (cadr p))
(define (procedure-body p) (caddr p))
(define (procedure-environment p) (cadddr p))
```

letの処理

26 行目

```
((let? exp) (eval (let->lambda exp) env))
```

202 行目

```
(define (let? exp) (tagged-list? exp 'let))
(define (let-variables exp) (map car (cadr exp)))
(define (let-exps exp) (map cadr (cadr exp)))
(define (let-body exp) (cddr exp))
(define (let->lambda exp)
  (cons
   (cons 'lambda
         (cons (let-variables exp) (let-body exp)))
   (let-exps exp)))
```

letもlambda式書き換えてから評価する．糖衣構文である．

環境の初期化

グローバル環境を作る.

308 行目

```
(define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                             (primitive-procedure-objects)
                             the-empty-environment)))
    (define-variable! 'true #t initial-env)
    (define-variable! 'false #f initial-env)
    initial-env))
(define the-global-environment (setup-environment))
```

さらに必要な関数を定義する

332 行目

```
(define (init-definitions)
  (let loop ((defs '((define (zero? x) (= x 0))
                     (define (cadr x) (car (cdr x)))
                     (define (cddr x) (cdr (cdr x)))
                     (define (cdar x) (cdr (car x)))
                     (define (caar x) (car (car x))))))
    (when (not (null? defs))
          (eval (car defs) the-global-environment)
          (loop (cdr defs)))))
(init-definitions)
```

REPLの処理

値の表示

323 行目

```
(define (user-print object)
  (if (compound-procedure? object)
      (display (list 'compound-procedure
                     (procedure-parameters object)
                     (procedure-body object)
                     '<procedure-env>))
      (display object)))
```

REPL本体

346 行目

```
(define (repl)
  (display "my-scheme> ")
  (let ((exp (read)))
    (when (not (eq? exp #!eof))
          (display "==> ")
          (user-print (eval exp the-global-environment))
          (newline)
          (repl))))
```

#!eofはファイルの終わりを表す．この場合，コンソールではControl-Dや Control-Zを表す

REPLの起動

355 行目

```
(display "** Scheme in Scheme **")
(newline)
(repl)
(display "Bye, see you.")
(newline))
```

REPLの動作 (1)

REPLに解釈させるためのプログラム map.scm

```
(define (mymap func lst)
  (if (null? lst) lst
      (cons (func (car lst))
            (mymap func (cdr lst)))))

(display mymap)
(let ((exp '(1 2 3 4 5 6)))
  (display (mymap (lambda (a) (* a a)) exp))(newline)
  (display (mymap (lambda (a) (cons a a)) exp))(newline))
```

```
OMacBook:yama517> kawa scheme.scm < map.scm
** Scheme in Scheme **
my-scheme> ==> (ok : mymap is defined)
my-scheme> ==> (compound-procedure (func lst)
                ((if (null? lst) lst
                  (cons (func (car lst)) (mymap func (cdr lst)))))
                <procedure-env>)
my-scheme> ==> (1 4 9 16 25 36)
((1 . 1) (2 . 2) (3 . 3) (4 . 4) (5 . 5) (6 . 6))

my-scheme> Bye, see you.
```

REPLの動作 (2)

REPLに解釈させるためのプログラム append.scm

```
(define (append lst1 lst2)
  (cond ((null? lst1) lst2)
        (else (cons (car lst1) (append (cdr lst1) lst2)))))

(display append)(newline)
(display (append '(1 2 3) '(a b c))) (newline)
```

```
OMacBook:yama521> kawa scheme.scm < append.scm
** Scheme in Scheme **
my-scheme> ==> (ok : append is defined)
my-scheme> ==> (compound-procedure (lst1 lst2)
      ((cond ((null? lst1) lst2)
                  (else (cons (car lst1) (append (cdr lst1) lst2)))))
      <procedure-env>)
my-scheme> ==>

my-scheme> ==> (1 2 3 a b c)
my-scheme> ==>

my-scheme> Bye, see you.
```

継続をコントロールする (1)

多くのプログラミング言語には「**継続 (continuation)**」という概念がある．継続は，実行されているプログラムの実行のある時点からプログラムの実行終了までの実行過程を抽象化したもののことである．Schemeでは以下のような `call-with-current-continuation` (`call/cc`) という関数によって現在実行している状態直後から実行終了までの過程を値として獲得することが可能である．

`call/cc`は一つの引数をとる関数を受け取り，その関数に`call/cc`自体の継続（`call/cc`が実行された直後からの継続）を値として渡して呼び出す．このとき，継続は関数であり，継続が実行されると実行位置が`call/cc`の直後となる．

継続をコントロールする (2)

以下に単純な例を示す.

```
(define call/cc call-with-current-continuation)

(define (test)
  (call/cc (lambda (ret)   ←call/cc自身の継続をもらってくる
             (let loop ((i 0))
               (display i)
               (if (>= i 10) (ret))
               (loop (+ i 1))))) ↑もらってきた継続を行使する
  (display "finished.")
  (newline))

(test)
```

call/ccの継続はここから後ろの実行

```
OMacBook:yama529> kawa test-callcc.scm
0 1 2 3 4 5 6 7 8 9 10finished.
OMacBook:yama530>
```

継続をコントロールする (3)

2重に呼び出された関数の内部から継続を実行しても問題ない.

```
(define call/cc call-with-current-continuation)

(define (test2)
  (call/cc (lambda (ret)
             (define (foo)
               (let loop ((i 0))
                 (display i)
                 (if (>= i 10) (ret))
                 (loop (+ i 1))))
             (foo)))
  (display "finished.")
  (newline))

(test2)
```

↑もらってきた継続をfooの内部で行使する

このように呼びされた内部から外へ脱出する動作は継続によって実現することができる. このような動作を大域脱出と呼ぶ.

Cにおける大域脱出

Cでは標準関数としてsetjmp, longjumpが用意されている．この２つの関数が利用するデータ構造として jmp_buf がある

```
#include <setjmp.h>
#include <stdio.h>

void foo(jmp_buf *jmp){
  printf("foo\n");
  longjmp(*jmp, 1);
  printf("bar\n");
}

int main(){
  jmp_buf jmp1;
  if (setjmp(jmp1) == 0){
    foo(&jmp1);
  } else {
    printf("finished\n");
  }
  return 0;
}
```

```
OMacBook:yama533> gcc c_longjmp.c -o c_longjmp
OMacBook:yama534> ./c_longjmp
foo
finished
OMacBook:yama535>
```

setjmpによって位置を決めて，longjmpを実行すると，setjmpの直後に再び返ってくる．左のコードの場合，値が0でなければ，”finished”を出力して停止する．

C++における大域脱出 (1)

C++ではtry-catchの構文を利用することができる.

```
#include <stdio.h>

void foo();

void foo(){
  printf("This is foo.\n");
  throw 3;
  printf("This is foo again.\n");
}
int main(){
  try {
    foo();
    foo();
  } catch(int i) {
    printf("finished (%d).\n", i);
  }
  return 0;
}
```

C++では整数でも文字列でもthrowできる.

C++における大域脱出 (2)

C++でも後述するJavaでも型によってcatchを複数定義することができる.

```
#include <stdio.h>

void foo();

void foo(){
  printf("This is foo.\n");
  throw "Hello";
  printf("This is foo again.\n");
}
int main(){
  try{
    foo();
  }catch(int i){
    printf("finished (%d).\n", i);
  }catch(char *message){
    printf("finished [%s].\n", message);
  }
  return 0;
}
```

C++でもJavaでもthrowする型によってcatchを選ぶことができる.

Javaにおける大域脱出

Javaではtry-catch-finally構文が使える．finallyはどのような状況でも必ず実行される．

```
public class GE{
  public static void foo() throws Exception{
    System.out.printf("This is foo.\n");
    // System.exit(1);
    if (1 == 1) throw new Exception();
    System.out.printf("This is foo again.\n");
  }
  public static void main(String [] args){
    try {
        foo();
        foo();
    } catch(Exception e){
        System.out.printf("finished (%s).\n", e.toString());
    } finally{
        System.out.printf("finally.\n");
    }
  }
}
```

強制的にexitを使うとfinallyは実行されない

Javaではthrowできるのは

Throwableなオブジェクトのみ

```
OMacBook:yama568> java GE
This is foo.
finished (java.lang.Exception).
finally.
```

継続を用いてジェネレータを作る

継続は色々に応用することができる．ジェネレータ (generator) は関数内部での状態を一時保存した状態で呼び出し側にもどり（その時返り値が戻る），後で再び保存した状態から実行できるしくみである．

```
(define call/cc call-with-current-continuation)

(define (count-generator)
  (let ((mycont '()))
    (lambda (yield)
      (if (null? mycont)
          (let loop ((i 0))
            (call/cc
             (lambda (cont)
               (set! mycont cont)
               (yield i)))
            (loop (+ i 1)))
          (mycont '())))))
```

このプログラムはkawaでは動かない．これはkawaの継続が完全な形で実現されていないからである．

この間が無限ループになっているが，途中で継続yieldを使って呼び出したところに戻る．戻る前に戻るための継続をmycontに保存しておく

```
OMacBook:yama525> gosh test-gen.scm
0
1
2
```

```
(define gen (count-generator))
(display (call/cc (lambda (yield) (gen yield))))(newline)
(display (call/cc (lambda (yield) (gen yield))))(newline)
(display (call/cc (lambda (yield) (gen yield))))(newline)
```

PythonとJavaScriptにおけるジェネレータ

PythonやJavaScriptでは普通にジェネレータが利用できる．ジェネレータを生成するには，関数のようにプログラムを書き，returnの代わりにyeildを用いる．

```
def count_generator():
    i = 0
    while True:
        yield i
        i = i + 1

gen = count_generator()

print gen.next()
print gen.next()
print gen.next()
```

ここで一時呼び出し元に戻る

```
function count_generator(){
    var i = 0;
    while (true){
        yield i;
        i = i + 1;
    }
}

var gen = count_generator();
print(gen.next());
print(gen.next());
print(gen.next());
```

```
OMacBook:yama529> python test-gen2.py
0
1
2
```

Python

```
OMacBook:yama535> js test-gen2.js
0
1
2
```

JavaScript

継続によるコルーチンのプログラミング (1)

いくつかの仕事を並列に行うためのしくみとして**コルーチン (coroutine)**がある．OS上のプロセスの並列と異なり，それぞれの関数がそれぞれの仕事を譲り合って計算が進む．

第5回の講義で説明したキューを利用する．キューに関する関数は `make-an-empty-queue, enqueue, dequeue` である．呼び出されるプロセスには以下のような関数で表現される．

```
(define (work1 pause)
  (define (work1-iter i)
    (display (list 'work1 i))
    (newline)
    (set! pause
          (call/cc
           (lambda (resume)
             (pause resume))))
    (work1-iter (+ i 1)))
  (work1-iter 0))
```

```
(define (work2 pause)
  (define (work2-iter i)
    (display (list 'work2 (* i i)))
    (newline)
    (set! pause
          (call/cc
           (lambda (resume)
             (pause resume))))
    (work2-iter (+ i 1)))
  (work2-iter 0))
```

それぞれの関数への引数pauseは仕事を中段するための継続である．pauseを実行するとき，関数への復帰のための継続を渡す

継続によるコルーチンのプログラミング (2)

それぞれの関数を順に呼び出す側のプログラムは以下のとおり.

```
(define myqueue (make-an-empty-queue))
(enqueue myqueue work1)
(enqueue myqueue work2)
(define (main-loop)
  (let ((proc (dequeue myqueue)))
    (let ((resume
           (call/cc
            (lambda (pause)
              (proc pause)))))
      (enqueue myqueue resume)))
  (main-loop))

(main-loop)
```

```
osami-2:yama510> gosh test-coroutine.scm
(work1 0)
(work2 0)
(work1 1)
(work2 1)
(work1 2)
(work2 4)
(work1 3)
(work2 9)
(work1 4)
(work2 16)
(work1 5)
(work2 25)
(work1 6)
(work2 36)
(work1 7)
(work2 49)
.....
```

キューから取り出して実行し，返って来た継続をキューに入れる．この場合，2つの関数は共に無限ループである.