

# 「プログラミング言語」

## SICP 第4章

### ～ 超言語的抽象～

### その5

五十嵐 淳

igarashi@kuis.kyoto-u.ac.jp

京都大学

July 14, 2015

# 今日のメニュー

## 4.3 への準備運動のつづき:

- 先週の宿題
- 4.2 $\frac{1}{2}$ : Variations on a Scheme – Exception handling
  - ▶ 4.2 $\frac{1}{2}$ .1: 例外 (実行時エラー) と例外処理機構
  - ▶ 4.2 $\frac{1}{2}$ .2: 継続
  - ▶ 4.2 $\frac{1}{2}$ .3: 継続渡しインタプリタ
  - ▶ 4.2 $\frac{1}{2}$ .4: catch/throw の実装

## 4.2<sup>1</sup>/<sub>2</sub>.2 の復習

### 継続とは

「計算プロセス・手続き的作業の  
(各時点における) 残りの計算, 残りの作業」

or

TODO リスト

- 例外処理は継続の操作と見なせる
- 継続をデータ化するインタプリタがあれば例外処理が実現できる

# 評価プロセスにおける継続

例:  $(+ (* e_1 e_2) e_3)$  の評価プロセス

式の形が関数適用だとわかった時点での継続

- 1  $(* e_1 e_2)$  の評価をする (値を  $v_1$  とする)
- 2  $e_3$  の評価をする (値を  $v_2$  とする)
- 3  $v_1$  と  $v_2$  の和を求める

# 評価プロセスにおける継続

例:  $(+ (* e_1 e_2) e_3)$  の評価プロセス

次の式も関数適用だとわかった時点での継続

- ①  ~~$(* e_1 e_2)$  の評価をする (値を  $v_1$  とする)~~
  - ①  $e_1$  の評価をする (値を  $v_{11}$  とする) .
  - ②  $e_2$  の評価をする (値を  $v_{12}$  とする) .
  - ③  $v_{11}$  と  $v_{12}$  の積を求める (値を  $v_1$  とする) .
- ②  $e_3$  の評価をする (値を  $v_2$  とする)
- ③  $v_1$  と  $v_2$  の和を求める

## 4.2<sup>1</sup>/<sub>2</sub>.3: 継続渡しインタプリタ

### 4.1 のインタプリタの主要関数

- eval:
  - ▶ 入力: 式と環境
  - ▶ 出力: 入力式の値
- apply:
  - ▶ 入力: 関数値と引数 (の値)
  - ▶ 出力: 適用結果の値

## 4.2<sup>1</sup>/<sub>2</sub>.3: 継続渡しシインタプリタ

### 継続渡しシインタプリタの主要関数

- eval:
  - ▶ 入力: 式と環境と継続
  - ▶ 出力: 入力式の値  
に対して継続が表現する操作をした結果の値
- apply:
  - ▶ 入力: 関数値と引数(の値)と継続
  - ▶ 出力: 適用結果の  
値に対して継続が表現する操作をした結果の値
- apply-cont:
  - ▶ 入力: 継続と値
  - ▶ 出力: 入力値に対して継続が表現する操作をした  
結果の値

# 継続渡し eval のデモ

```
(eval '(cons 1 2) the-global-environment  
      (make-haltc))
```

;; make-haltc: TODO リストの末尾を表す継続を作る

```
(eval '(define x (cons 1 2))  
      the-global-environment  
      (make-haltc))
```

```
(eval 'x the-global-environment (make-haltc))
```

# 継続渡し eval のデモ

```
(eval '(cons 1 2) the-global-environment
      (make-definec 'y the-global-environment
                    (make-haltc)))
```

;; make-definec:

;; 「定義をする」を表す継続を作る関数

;; 作られた継続:

;; ・ y を式の値として定義

;; ・ (おわり)

```
(eval 'y the-global-environment (make-haltc))
```

# 継続渡し eval のデモ

```
(eval '(define z (- (+ 1 2) (+ 4 4)))  
      the-global-environment  
      (make-haltc)))
```

```
(eval '(- (+ 1 2) (+ 4 4))  
      the-global-environment  
      (make-definec 'z the-global-environment  
                    (make-haltc)))
```

# 継続渡し eval のデモ

```
;; "-" の評価が終わった時点では...  
(list-of-values '((+ 1 2) (+ 4 4))  
  the-global-environment  
  (make-aplyc (list 'primitive -)  
    (make-definec 'y the-global-environment  
      (make-haltc))))  
;; list-of-value に与えた継続:  
;;   • "-" を list-of-values の結果に適用し,  
;;   • その値を y として定義し,  
;;   • (おわり)
```

## 継続 as 穴の空いた式

継続は「穴 (以下 [] で表す) の空いた式」(穴に値が埋まると計算が発生する) と思うとよいかも。

- `definec` (define <変数> [])
- `applyc` (<関数> [])
- `haltc` []

# 継続の合成

- `make-XXXc` の最後の引数は, `XXXc` の後の動作を示す継続
- なので...

```
(make-applyc (list 'primitive -)
  (make-definec 'y the-global-environment
    (make-haltc))))
```

は `(define y (- []))` 相当

- ▶ `(- [])` を `(define y [])` でくるんだ (後者の穴に前者をはめこんだ)

# 継続渡し eval のデモ

```
(eval '(define y (throw 'a 2))  
  the-global-environment  
  (make-haltc)))
```

```
(eval '(throw 'a 2) the-global-environment  
  (make-definec 'y the-global-environment  
  (make-haltc)))
```

# eval と apply-cont の関係

相互に呼び合う仲:

- eval は, 自分の仕事が終わったら (式の値が得られたら) apply-cont を呼ぶ
- apply-cont は, 継続中に「~を評価する」という作業をこなすために eval を呼ぶ

# eval の定義

引数として継続を表す `cont` を追加

```
(define (eval exp env cont)
  (cond
    ((self-evaluating? exp) ...)
    ((variable? exp) ...)
    ...
  ))
```

## eval の定義

式からすぐに値が出る場合 (定数, 変数, 引用, ラムダ) は, `apply-cont` を呼んで値を継続 (cont) に渡す

```
(define (eval exp env cont)
  (cond
    ((self-evaluating? exp)
     (apply-cont cont exp))
    ((variable? exp)
     (apply-cont cont
                  (lookup-variable-value exp env)))
    ((quoted? exp)
     (apply-cont cont
                  (text-of-quotation exp))))
```

# 補助関数 eval-if

## 4.1 のインタプリタ:

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

青字部分が条件部 (if-predicate) を評価した後の計算, つまり継続

# 継続渡しシインタプリタの eval-if

```
(define (eval-if exp env cont)
  (eval (if-predicate exp) env
        (make-testc (if-consequent exp)
                     (if-alternative exp)
                     env cont)))
```

- `make-testc`: 前頁青字部分を表す継続を作る関数
  - ▶ ここで作られた継続は，条件部の値といっしょに `apply-cont` に渡される．
  - ▶ 関連する関数群: `testc?`, `testc-true`, `testc-false`, `testc-env`, `testc-cont`

# apply-cont の定義

```
(define (apply-cont cont val)
  (cond
    ((testc? cont) ;; make-testc で作られた?
     ;; 継続から残りの計算に使うデータを取り出す
     (let ((consequent (testc-true cont))
           (alternative (testc-false cont))
           (env (testc-env cont))
           (cont' (testc-cont cont)))
       ;; 4.1 の eval-if の青字部分!
       (if (true? val)
           (eval consequent env cont')
           (eval alternative env cont'))
       ...)))
```

# 継続渡しインタプリタのポイント

- 値を返す時は `apply-cont` を呼ぶ
- ひとつめの `eval` の再帰呼出し以降の残りの計算を継続化 (`make-XXXc` 関数)
- 継続が表す実際の処理は `apply-cont` に記述

## eval-sequence の場合

元の (4.1 の) 定義:

```
(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))))
```

## 継続渡し eval-sequence

```
(define (eval-sequence exps env cont)
  (cond ...
    (else
     (eval (first-exp exps) env
           (make-begin
            (rest-exps exps) env cont))))))
```

;; apply-cont の対応する条件節の抜粋

```
((begin? cont)
 (eval-sequence (begin-rest-exps cont)
                (begin-env cont)
                (begin-cont cont)))
```

# eval-assignment の場合

元の (4.1 の) 定義:

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

## 継続渡し eval-assignment

```
(define (eval-assignment exp env cont)
  (eval (assignment-value exp) env
        (make-assignment
         (assignment-variable exp) env cont)))
```

;; apply-cont の対応する条件節の抜粋

```
((assignment? cont)
 (set-variable-value!
  (assignment-var cont)
  val (assignment-env cont))
;; set! の戻り値 'ok を継続に渡す
(apply-cont (assignment-cont cont) 'ok))
```

# 関数適用

## 4.1 のインタプリタ:

```
(define (eval exp env)
  (cond ...
    ((application? exp)
     (my-apply (eval (operator exp) env)
                (list-of-values (operands exp) env))))
```

## 継続の表す作業 (2 段階):

- ① 引数式を評価した結果のリストを作って
- ② それを関数といっしょに apply に渡す

# 継続渡しバージョン

```
(define (eval exp env cont)
  (cond ...
    ((application? exp)
     (eval (operator exp) env
           (make-operandsc
            (operands exp) env cont))))
```

;; apply-cont の対応する条件節の抜粋

```
((operandsc? cont)
 (list-of-values (operandsc-exps cont)
                 (operandsc-env cont)
                 (make-applyc val (operandsc-cont cont))))
```

make-operandsc で作られた継続が表す作業:

- ① 引数式を評価した結果のリストを作って
- ② それを関数といっしょに apply に渡す

2番目の作業は make-applyc で継続化されている

;; apply-cont の対応する条件節の抜粋

```
((apply? cont)
  ;; それ(引数リスト)を関数とともに apply に渡す
  (my-apply (applyc-proc cont)
            val
            (applyc-cont cont)))
```

# list-of-values

「式のリストからそれを評価した値のリストを作る」  
作業の分割

- ① 最初の式を評価
- ② 残りの式リストを評価
- ③ それらを `cons` でつなぐ

```
(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons
        (eval (first-operand exps) env)
        (list-of-values
          (rest-operands exps) env))))
```

## 継続渡し list-of-values

```
(define (list-of-values exps env cont)
  (if (no-operands? exps)
      (apply-cont cont '())
      (eval (first-operand exps) env
            (make-restopsc (rest-operands exps)
                           env cont))))
```

## :: apply-cont の対応する条件節の抜粋

```
((restopsc? cont) ;; 2. 残りの式リストを評価
 (list-of-values
  (restopsc-rest cont)
  (restopsc-env cont)
  (make-cons val (restopsc-cont cont))))
((consc? cont) ;; 3. それらを cons でつなぐ
 (apply-cont (consc-cont cont)
              (cons (consc-val cont) val)))
```

# 継続の種類のもつめ(1)

testc	条件判定をして、適当な式の評価を行う (if [] <式1> <式2>)
assignc	変数への代入を行い、'ok を返す (set! <変数> [])
definec	変数定義を行い、'ok を返す (define <変数> [])
beginc	begin の残りの式の評価を行う (begin [] ...)
operandsc	関数適用の引数部を順次評価して、 apply を呼び出し関数適用を行う (<関数> <値> ... [] <式> ...)

# 継続の種類のもつめ (1)

- `applyc` 関数を適用する  
(`operandsc` が表す計算の最後の部分)  
(`<関数> <値> ... []`)
- `restopsc` 残りの引数を評価し, 最初の引数の値と  
`cons` で繋ぐ  
(`cons [] <式>`)
- `consc` 最初の引数の値と残りの引数の値を  
`cons` で繋ぐ  
(`cons <値> []`)
- `haltc` 何もしない (継続リストの末尾)

## 4.2<sup>1</sup>/<sub>2</sub>.4: catch/throw の実装

例外処理 継続に対する操作

- error の実行 TODO リストを捨てる
- throw の実行 最も近くタグが等しい catch までの継続を捨てる

# catch の実装

- ① タグ部の評価
- ② 継続に印をつけて本体の評価

```
(define (eval exp env cont)
  (cond ...
    ((catch? exp)
     (eval (catch-tag exp) env
           (make-cabodyc
            (catch-body exp) env cont))))
```

```

(define (apply-cont cont val)
  (cond ...
    ((cabodyc? cont)
     (eval-sequence
      (cabodyc-body cont)
      (cabodyc-env cont)
      (make-catchc val ;; catch の印
        (cabodyc-cont cont))))
    ;; throw がなく本体の評価が終了
    ((catchc? cont)
     (apply-cont (catchc-cont cont) val)))

```

# throw の実装

- ① タグ式の評価
- ② ふたつめの式 (投げられる値となる) の評価
- ③ 最も近い catch の印 (catchc) までの継続を捨てて、残りの継続に投げられた値を渡す

```
(define (eval exp env cont)
  (cond ...
    ((throw? exp)
     (eval (throw-tag exp) env
           (make-thbodyc (throw-body exp)
                        env cont))))
```

```
(define (apply-cont cont val)
  (cond ...
    ((thbodyc? cont)
     (eval
      (thbodyc-body cont) (thbodyc-env cont)
      (make-throwc val (thbodyc-cont cont))))
    ...))
```

```

(define (apply-cont cont val)
  (cond ...
    ((throwc? cont)
     (let ((stripped-cont
            (first-matching-catch
             (throwc-tag cont)
             (throwc-cont cont))))
       (if stripped-cont
           ;; catch が見つかったら評価再開
           (apply-cont stripped-cont val)
           ;; uncaught exception を示す評価結果
           (list 'uncaught
                  (throwc-tag cont) val))))))

```

```

(define (first-matching-catch thrown-tag cont)
  (define (loop cont)
    (cond
      ((haltc? cont) false)
      ;; catch の印ではない継続の場合
      ((testc? cont) (loop (testc-cont cont)))
      ...
      ((catchc? cont) ;; catch の印だ!
       ;; タグは等しい?
       (if (eq? thrown-tag (catchc-tag cont))
           (catchc-cont cont)
           (loop (catchc-cont cont))))
      ...))
  (loop cont))

```

# 宿題： 7/21 16:30 締切

- 継続渡し evaluator に and を (derived-expression とせず, eval-and 関数を追加する形で) 追加せよ。(オプション: 練習問題 4)
- レポートには
  - ▶ 考え方の説明
  - ▶ プログラムリストと考え方の対応
  - ▶ 実行例を示すこと
- レポート (pdf) とプログラムファイルを提出
- 友達に教えてもらったなら、その人の名前を明記
- web は出典を明記(「同じ」回答は減点)

## 練習問題4の補足説明

`(catch tag handler exp1 ... expn)`

- `tag` を評価する (値を `t` とする)
- `expi` を順に評価する
- 評価が正常に終われば `expn` の値が `catch` 全体の値
- 例外 (`throw 't v`) が発生した  $\implies$  関数適用 (`handler v`) を評価した結果が `catch` 全体の結果

# 例

```
> (catch 'a (lambda (x) (+ x 1))  
      (+ 2 (throw 'a 4)))
```

5