

# 「プログラミング言語」

## SICP 第4章

### ～ 超言語的抽象～

### その7

五十嵐 淳

igarashi@kuis.kyoto-u.ac.jp

京都大学

July 25, 2012

# 今日のメニュー

- 4.3: Variations on a Scheme – Non-deterministic Computing
  - ▶ 4.3.1: 特殊形式 `amb` と探索
  - ▶ 4.3.2: 非決定的プログラムの例
  - ▶ 4.3.3: `amb` インタプリタの実装

# Scheme 変奏曲 – 非決定的計算

非決定的計算  $\stackrel{\text{def}}{=}$  結果が複数あるような計算

- 計算過程の多世界 (パラレルワールド) 解釈?
- “generate and test” 方式で (探索) 問題を解くのに向いている

⇒ リスト, ストリームを使って散々(?) やったこと

## 4.3.1: 特殊形式 amb と探索

特殊形式 amb

$$(\text{amb } e_1 \dots e_n)$$

- $e_i$  を非決定的に選択し評価する
- 結果は  $n$  通り
- $n = 0$  でもよい!
  - ▶ 計算結果がない場合 (探索の失敗など) を表す

# amb と catch/throw の比較

- 類似点

- ▶ どちらも実行の「ジャンプ」を引き起こす

- ★ amb は catch と throw 両方の役割をする

- 相違点

- ▶ catch が働く (catch する) 有効範囲は本体式の実行中
- ▶ amb の有効範囲は残りの計算全部

# amb は catch と throw 両方の役割をする

- (throw ...) ~ (amb)
- (amb e1 e2): e1 の評価中に (amb) が実行されると、その評価が中断し e2 の評価から再開する

(amb (+ 2 (amb) 3) 4)

⇒ 4

- ▶ (amb) のところにふつうの式があれば、e1 の値がそのまま全体の値

## catch と amb の有効範囲の比較

```
(let ((x (catch 'a 2)))  
  (if (even? x) (throw 'a true) x))  
⇒ uncaught exception
```

```
(let ((x (amb 2 3)))  
  (if (even? x) (amb) x))  
⇒ 3
```

(amb ...) を抜けた後の (amb) の実行は、実行の**巻き戻し** (backtrack という) を起こす!

# 継続の言葉でいうと...

- `throw` は継続の一部を飛ばす ⇒ 「早送り」
- `(amb)` は「早送り」することもあるが、既に終了して `TODO` リストから消された作業を復活させて「巻き戻す」ことも
  - ▶ 常に「最後に `amb` に実行が突入した時点での継続」が実行される

# 継続渡しインタプリタによる amb の実装

アイデア: 二種類の継続を引数とする eval

- 式の評価がふつうに終了した時にその値を渡す継続
- 式の評価中に (amb) が実行された時に呼び出す継続 fcont (failure continuation)
  - ▶ 「最後に amb に実行が突入した時点での継続」
  - ▶ amb に関係のない部分では, 引数として受け取った fcont はそのまま次の処理に渡される
- eval の返り値:
  - ▶ 式の値 (ひとつめの答え)
  - ▶ “no-more-values” (答えなし)
  - ▶ “no-problem” (いきなり try-again を入力した)

# eval/apply/apply-cont の定義

引数として, (amb) 実行後に実行を再開するための継続 **fcont** (failure continuation) を追加

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

```
(define (my-apply proc args cont fcont)  
  ...)
```

```
(define (apply-cont proc args cont fcont)  
  ...)
```

## eval: 式からすぐに値が出る場合

- 前と同じく `apply-cont` を呼んで値を継続 (cont) に渡す
- `fcont` はそのまま

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

## その他の場合の代表: if

- 前との違いは `fcont` の部分だけ
- `fcont` はもらったものをそのまま次の処理 (`eval`) に渡す

```
;; eval から抜粋
```

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

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

## amb の処理

```
(define (amb? exp) (eq? (car exp) 'amb))
(define (amb-choices exp) (cdr exp))

(define (eval exp env cont fcont)
  (cond ...
    ((amb? exp)
     (eval-amb (amb-choices exp)
               env cont fcont))
    ...
  ))
```

## eval-amb

- 選択肢がなければ, `apply-fcont` を使って `fcont` を呼び出す
- 選択肢があれば
  - ▶ 最初の式の評価に入る
  - ▶ `make-ambc` で, 残りの選択肢とそれを評価するための情報を含む新しい `fcont` を作る

```
(define (eval-amb choices env cont fcont)
  (if (null? choices)
      (apply-fcont fcont)
      (eval (car choices) env cont
            (make-ambc (cdr choices)
                       env cont fcont))))
```

# fcont の表現

- ambc: 「一番最後に突入」した amb の継続を表す
- initf1c: 空の継続 ⇒ これ以上戻れるところはない
  - ▶ 当初入力された式が記録されている
- initf2c
- revassignc

# apply-fcont

fcont の挙動を決める関数

- `ambc` の場合，残りの選択肢の実行のやり直し
  - ▶ 残りの選択肢が空なら，その前の `amb` まで戻る
- `initf1c` の場合，“no-more-val” を返す

```
(define (apply-fcont fcont)
  (cond
    ((ambc? fcont)
     (eval-amb
      (ambc-exps fcont) (ambc-env fcont)
      (ambc-cont fcont) (ambc-fcont fcont)))
    ((initf1c? fcont)
     (list 'no-more-val (initf1c-exp fcont)))
    ...))
```

# REPL の実装: try-again を理解する

- try-again を入力 ⇒ 前に入力された式の最後の amb 分岐まで戻る

# REPL の実装: try-again を理解する

- try-again を入力 ⇒ 前に入力された式の最後の amb 分岐まで戻る
- どうやって？

# REPL の実装: try-again を理解する

- try-again を入力 ⇒ 前に入力された式の最後の amb 分岐まで戻る
- どうやって?
- 式の評価結果を「式の値と最後の amb 分岐で作られた fcont の組」とすればよい!

# REPL の実装: try-again を理解する

- try-again を入力 ⇒ 前に入力された式の最後の amb 分岐まで戻る
- どうやって?
- 式の評価結果を「式の値と最後の amb 分岐で作られた fcont の組」とすればよい!

```
(define (apply-cont cont val fcont)
  (cond ((haltc? cont)
        (list 'normal val fcont))
        ;; 次の入力が try-again だったら
        ;; この fcont を呼ぶ!
        ...))
```

## REPL の実装 (2)

```
(define (driver-loop)
  (define (loop fcont) ;; REPL の実体
    ;; 1. 入力受付

    ;; 2. 入力が try-again なら fcont を呼ぶ

    ;; 3. 入力が式なら評価
    ;; 4. 出力の種類によって場合わけ
  )
  (loop ...))
```

## REPL の実装 (2)

```
(define (driver-loop)
  (define (loop fcont) ;; REPL の実体
    ;; 1. 入力受付
    (prompt-for-input input-prompt)
    (let* ((input (read))
           ;; 2. 入力が try-again なら fcont を呼ぶ
           ;; 3. 入力が式なら評価
           ;; 4. 出力の種類によって場合わけ
           )
      (loop ...)))
```

## REPL の実装 (2)

```
(define (driver-loop)
  (define (loop fcont) ;; REPL の実体
    ;; 1. 入力受付
    (prompt-for-input input-prompt)
    (let* ((input (read)))
      ;; 2. 入力が try-again なら fcont を呼ぶ
      (output
       (if (eq? input 'try-again)
           (apply-fcont fcont)
           ;; 3. 入力が式なら評価
           ;; 4. 出力の種類によって場合わけ
           ))
      (loop fcont))
    )
  (loop ...))
```

## REPL の実装 (2)

```
(if (eq? input 'try-again) ...  
;; 3. 式ならば fcont を空 (initf1) にして評価  
  (begin  
    (newline)  
    (display ";;; Starting a new problem")  
    (eval input the-global-environment  
          (make-haltc)  
          (make-initf1c input))))))  
;; 4. 出力の種類によって場合わけ  
...  
...
```

## REPL の実装 (2)

```
(let* ((input (read))  
      (output ...))
```

;; 4. 出力の種類によって場合わけ

```
(cond ((tagged-list? output 'normal)  
      ;; 正常終了: (normal 値 fcont)  
      (announce-output output-prompt)  
      (user-print (cadr output))  
      (loop (caddr output))))
```

## REPL の実装 (2)

```
((tagged-list? output 'no-more-val)
;; 答がない(尽きた): (no-more-val 式)
  (announce-output
   ";;; There are no more values of ")
  (user-print (cadr output))
  (loop ...))
```

# (loop ...) の ... について

- 最初に driver-loop を呼び出した時
- 答が尽きた直後

これは try-again をしてはいけない (問題が設定されていない) ところ!



fcont の表現

- ambc: 「一番最後に突入」した amb の継続を表す
- initf1c: 空の継続 ⇒ これ以上戻れるところはない
- **initf2c: 問題が設定されていない時に使う継続**
- revassignc

# apply-fcont, driver-loop 再掲

```
(define (apply-fcont fcont)
  (cond
    ((initf2c? fcont) (list 'no-problem))
    ((ambc? fcont) ...)
    ((initf1c? fcont) ...)
    ...))
```

```

(define (driver-loop)
  (define (loop fcont)
    (...
      (cond ...
        ((tagged-list? output 'no-more-val)
         ... ;; 答えがない(尽きた)
         (loop (make-initf2c))))
        ((tagged-list? output 'no-problem)
         ;; 問題未設定
         (announce-output
          ";;; There is no current problem")
         (loop (make-initf2c)))
        (loop (make-initf2c))))))

```

# 一旦まとめ

- eval

- ▶ 入力: 式, 環境, 継続, 失敗継続の4つ
- ▶ 出力: 以下の三通りのいずれか
  - ★ 'normal と値と失敗継続の組
  - ★ 'no-more-vals と入力式
  - ★ 'no-problem
- ▶ fcont をいじるのは入力式が amb の場合だけ

- fcont の表現

- ▶ ambc: 「一番最後に突入」した amb の継続を表す
- ▶ initf1c: 空の継続 ⇒ これ以上戻るところ無し
- ▶ initf2c: 問題が設定されていない時に使う継続
- ▶ revassignc

## さらにもう一工夫: set! と amb

```
(let ((a 1))  
  (set! a (+ a (amb 100 200 300))))  
a)
```

- 最初の実行の値は？

# さらにもう一工夫: set! と amb

```
(let ((a 1))  
  (set! a (+ a (amb 100 200 300))))  
a)
```

- 最初の実行の値は?  $\Rightarrow$  101
- try-again した時の値は?

# さらにもう一工夫: set! と amb

```
(let ((a 1))  
  (set! a (+ a (amb 100 200 300)))  
  a)
```

- 最初の実行の値は?  $\Rightarrow$  101
- try-again した時の値は?  $\Rightarrow$  201
- try-again した時の fcont の
  - ▶ (ambc-exps fcont) は?
  - ▶ (ambc-env fcont) の a の値は?

## さらにもう一工夫: set! と amb

```
(let ((a 1))  
  (set! a (+ a (amb 100 200 300))))  
a)
```

- 最初の実行の値は?  $\Rightarrow$  101
- try-again した時の値は?  $\Rightarrow$  201
- try-again した時の fcont の
  - ▶ (ambc-exps fcont) は?  $\Rightarrow$  (200 300)
  - ▶ (ambc-env fcont) の a の値は?  $\Rightarrow$  1

$\Rightarrow$  backtrack する時には a の値を戻す必要あり!

# apply-cont 中の代入を扱う部分

```
(define (apply-cont cont val fcont)
  (cond ...
    ((assignc? cont)
     (let* ((var (assignc-var cont))
            (env (assignc-env cont))
            (oldval
              (lookup-variable-value var env)))
       (set-variable-value! var val env)
       (apply-cont (assignc-cont cont) 'ok
                   ;; 代入を「なかったことにする」ための
                   ;; 情報を fcont に記録
                   (make-revassignc
                    var oldval env fcont))))))
```

# 代入を「なかったことにする」

```
(define (apply-fcont fcont)
  (cond ...
    ((revassignc? fcont)
     ;; 更新前の値の代入による打ち消し
     (set-variable-value!
      (revassignc-var fcont)
      (revassignc-old fcont)
      (revassignc-env fcont)))
     ;; さらに巻き戻す
     (apply-fcont (revassignc-fcont fcont))))
```

# まとめ

- eval

- ▶ 入力: 式, 環境, 継続, 失敗継続の4つ
- ▶ 出力: 以下の三通りのいずれか
  - ★ 'normal と値と失敗継続の組
  - ★ 'no-more-vals と入力式
  - ★ 'no-problem

- ▶ fcont をいじるのは入力式が amb の場合だけ

- fcont の表現

- ▶ ambc: 「一番最後に突入」した amb の継続を表す
- ▶ initf1c: 空の継続 ⇒ これ以上戻るところ無し
- ▶ initf2c: 問題が設定されていない時に使う継続
- ▶ revassignc: 代入を打ち消すための処理を表す
  - ★ 内部定義の define は set! に変換する必要あり

# 教科書の amb インタプリタ

- 4.1.7 のテクニックをあまり意味もなく採用している
- 継続を，シンボルとリストではなく，関数値 (`lambda`) で表現している (関数化変換)

# 継続の関数化

継続はある意味関数のようなもの:

- `c` が `haltc` の時 `apply-cont` の定義より, この継続は

```
(lambda (val fcont) (list 'normal val fcont))
```

のようなもの

⇒ `(make-haltc)` を上の `lambda` で置き換えてよいのでは?

- c が testc? の時

```
(lambda (val fcont)
  (if (true? val)
      (eval (testc-true cont) ... fcont)
      (eval (testc-false cont) ... fcont)))
```

のように働く

⇒ **赤字部分**は make-testc の引数なので,

```
(make-testc e1 e2 env cont) は
(lambda (val fcont)
  (if (true? val)
      (eval exp1 env cont fcont)
      (eval exp2 env cont fcont)))
```

で置き換えられる？

# 関数化変換の一般的なレシピ

- ① (make-XXXc ...) を, 対応する apply-cont の動作を表す lambda で置き換える
  - ② (apply-cont c ...) を, (c ...) と c の直接呼出で置き換える
- 一般のデータ構造に使えるテクニック
  - データ構造を「使う」(解釈する)関数がひとつだけである必要がある
    - ▶ amb インタプリタの場合: 継続は apply-cont , 失敗継続は apply-fcont
- ⇒ catch/throw インタプリタには使えない  
first-matching-catch がある

# 今週の宿題

...はありません .

# 試験について

- 試験範囲:
  - ▶ 教科書 4 章 (4.1 ~ 4.3)
  - ▶ catch/throw についての補足資料
  - ▶ ただし 4.3.3 は講義内容を元にする
- 持ち込みは無のだが，インタプリタのプログラムを暗記する必要はない