情報システム科学実習 II (12)

継続と例外処理

1999 年 1 月 19 日 増原 英彦

継続 (**continuation**) は、(いま処理中の計算に対して)「残りの計算全て」という概念である。これを用いることで、例外処理などが簡潔に記述できるようになる。

1 n 個の値を入力する

例外的な処理は、現在処理中の動作とは別の処理を行う必要があるため、プログラムが複雑になる。「n 個の値を入力する」という問題を例にとって考えてみよう。

対話的に値を入力するためのクラスとして<InputArea>があるとする。(Web ページinput-area.scm に定義あり。)

```
STk> (define i (make <InputArea>))
i
STk> (input i "Value: ")
                  (ここで画面に入力用の領域が現われる。
                  Enter ボタンが押されると先へ進む。)
"Hello."
STk>
```

これを使って n 個の値を入力するプログラムを作ろう。まずはリストを作る一般的な関数make-list を定義する。この関数は(make-list gen num) のように使われ、i 番目の要素は(gen i) であるような長さnum のリストを作る (Web ページsimple-input-values.scm に定義あり。):

```
;;; i ~ (num-items − 1) 要素を作る
(define (make-list-from generator num-items i)
  (if (= i num-items)
      '()
      (let ((first (generator i))) ; i 番目の要素を作る
        (cons first
              ;; i+1 番目以降のリストを作る
              (make-list-from generator num-items
                              (+ i 1))))))

;;; 長さ num-items のリストを generator を使って作る
(define (make-list generator num-items)
  (make-list-from generator num-items 0))
```

使用例:

```
STk> (define (square x) (* x x)) ; 二乗
STk> (make-list square 10)       ; 0 ~ 9 を二乗した数のリスト
(0 1 4 9 16 25 36 49 64 81)
```

次に関数make-list とクラス<Input>を組み合わせてnum 個の値を対話的に入力する関数input-values を定義する。ここでは i 番目の値を入力するときに “Please enter value #i:” と聞くことにする。

```
;;; index 番目の値を入力する際のプロンプトを作る
(define (make-prompt index)
  (string-append
   "Please enter value #" (number->string index) ": "))

;;; num-values 個の値を入力する
(define (input-values num)
  (let ((input-area (make <InputArea>)))
    (make-list (lambda (index)
                 (input input-area (make-prompt index)))
               num)))
```

使用例 (実際に動かしてみて欲しい)。

2 素朴な例外処理

入力領域には “Cancel” ボタンが付いている。これをクリックすると、input は#f を返す。そこで、“Cancel” がクリックされたときに最初から値を入力しなおすようにしよう。

素朴なやり方として、「キャンセル」か「値」かを区別できるような抽象データを使う方法がある。ここでは、以下のような外部インターフェイスを持つ抽象データを使うとしよう。(Web ページcancel.scm に定義あり。)

cancel-value	「キャンセル」を表わす定数
(make-value value)	「値」を作る構築子
(cancel? cancel-or-value)	「キャンセル」かどうかを調べる
(get-value cancel-or-value)	「値」をとり出す選択子

これを使ってキャンセルに対応したプログラムを作るには、途中でキャンセルする可能性のある関数の返り値を以下のようにする:

- 正常に実行された場合は「値」を作って返す
- キャンセルされた場合は、「キャンセル」を返す

関数を呼び出す側は、答えが「値」か「キャンセル」かを判断し、「値」の場合は計算を続行し、「キャンセル」の場合は例外処理をする。

もとのプログラムは下のような呼び出し関係にあった:

```
input-values — n 個の値を入力
        make-list — n 要素のリストを作る
                make-list-from — i 番目からリストを作る
                        input(generator) — 値の入力
```

実際にキャンセルが起こるのは、メソッドinput の中である。そこでinput からの答えが#f であったら、「キャンセル」を返し、そうでなければ「値」を作って返すことにする。(関数名には、区別のためにnaive-をつけている。Web ページnaive-input-values.scm に定義あり。)

```
;; 「値」か「キャンセル」を返す input
(define (naive-input input-area prompt)
  (let ((value (input input-area prompt)))
    (if value                          ; キャンセル時は#f
        (make-value value)             ;「値」を作る
        cancel-value)))                ;「キャンセル」
```

関数make-list-from は、i 番目の値をgenerator (実際にはinput を呼ぶ関数クロージャ) を使って作る部分と、$i+1$ 番目以降のリストを作る部分がある。それぞれの中でキャンセルされる可能性があるので、そのときは、「キャンセル」値を返す。そうでないときは「値」から中身をとり出した上でcons をし、その結果を「値」にして返す。

```
;; リストの「値」か「キャンセル」を返す make-list-from
(define (naive-make-list-from generator num-items i)
  (if (= i num-items)
      (make-value '())             ; 空リストを「値」にして返す
      (let ((first (generator i))); i 番目の要素を作る
        (if (not (cancel? first)) ; 「値」ならば
            ;; i+1 番目以降のリストを作る
            (let ((rest (naive-make-list-from
                         generator num-items (+ i 1))))
              (if (not (cancel? rest))     ;「値」ならば
                  (make-value              ; cons したものを
                   (cons (get-value first) ; 「値」にして返す
                         (get-value rest)))
                  cancel-value));i+1 番目以降でキャンセル
            cancel-value))))    ;i 番目を作るときにキャンセル
```

関数make-list は単にmake-list-from を呼ぶだけ:

```
(define (naive-make-list generator num-items)
  (naive-make-list-from generator num-items 0))
```

関数input-values は、make-list から受け取った答えが「キャンセル」であれば入力をし直すようにする。

```
(define (naive-input-values num-values)
  (let* ((input-area (make <InputArea>))
         (values
          (naive-make-list                 ; num-values 個の
           (lambda (index)                 ; リストを作る
             (naive-input input-area (make-prompt index)))
           num-values)))
    (if (not (cancel? values))             ;「値」ならば
        (get-value values)                 ; 中身をとり出す
        ;;「キャンセル」ならば全体を再実行
        (naive-input-values num-values))))
```

このやり方では、ほとんど全ての関数が「値」あるいは「キャンセル」を返すようになるため、関数呼出の度に結果が「キャンセル」かどうかを調べたり、中身をとり出す判断を書かなければならなくなる。

実際にやりたいことは、input の最中にキャンセルされたら、(1) 現在実行中の処理を捨てて (2) 全体を再実行したい、だけである。しかし、関数的なプログラミングスタイル

では、現在の実行中の処理を捨てることが難しいため、このように複雑な記述になってしまうわけである。

参考: C 言語のようにgoto 文がある言語では、同じ関数中であれば「現在実行中の処理を捨てる」ことは比較的簡単にできる。しかし、C 言語の goto 文でも関数呼出があった場合には同じ問題が生じる。

3 継続渡し形式

継続 (continuation) とは、「その後の計算全て」を表わすようなデータである。継続を関数として表わし、関数呼び出しの際の引数として明示的に渡してゆくようなスタイルを継続渡し形式 (cotinuation-passing style) という。

3.1 簡単な例

下のようなプログラムを例にして考える:

```
(define (square z) (* x x)
(define (distance x y) (sqrt (+ (square x) (square y))))
```

関数distnace の中でsquare が 2 回呼び出されている。(square x) の呼び出しでは、「その後の計算」とは(sqrt (+ □ (square y))) になる。(□は(square x) の計算結果とする。) そこで、これを(lambda (X) (sqrt (+ X (square y)))) のようにして関数として表わすことにすると、以下のように書き直せる。(区別のために関数名にはCPS-を付けている。)

```
(define (CPS-square z cont) (cont (* x x))
(define (CPS-distance x y cont)
 (CPS-square x
  (lambda (X)
    (CPS-square y
      (lambda (Y)
        (cont (sqrt (+ X Y))))))))
```

3.2 変換規則

このような書き換えを CPS 変換という。CPS 変換の規則の概略は以下のようなものである:

- (define (f x y ...) e) という関数定義は、新しい引数cont と、e を継続cont によって変換した e' を用いて、(define (f x y ... cont) e') となる。
- 定数 c, 変数 v を継続 k によって変換したものはそれぞれ(k c), (k v)。
- 組み込み関数への適用 (o e_1 e_2) を継続 k によって変換したものは、e_2 を継続(lambda (v2) (k (o v1 v2)) によって変換して e_2' を作り、e_1 を継続(lambda (v1) e_2') によって変換したものになる。

- 関数への適用 $(e_0\ e_1)$ を継続 k によって変換したものは、e_1 を継続(lambda (v1) (v0 v1 k)) によって変換した e_1' を作り、e_0 を継続(lambda (v0) e_1') によって変換したものになる。

3.3 CPS によるの「n 個の値の入力」

最初に出てきた「n 個の値を入力する関数」を CPS に書き直すと次のようになる (区別のためにCPS-をつけている。また、実際に変換したものを見やすいように簡略化してある。Web ページcps.scm に定義あり。):

```
;; 継続を引数に加える
(define (CPS-input-values num-values k1)
  (let ((input-area (make <InputArea>)))
    (CPS-make-list
      (lambda (index k2)
        (CPS-input input-area (make-prompt index) k2))
      num-values k1)))

;; input は (ここでは) 組み込み関数なので適用した結果を継続に渡す
(define (CPS-input input-area prompt k)
  (k (input input-area prompt)))

;; 継続を引数に加える
(define (CPS-make-list generator num-items k)
  (CPS-make-list-from generator num-items 0 k))

(define (CPS-make-list-from generator num-items i k)
  (if (= i num-items)
      (k '())              ; 定数を継続に返す
      (generator i         ; i 番目の要素を作る
        (lambda (v1)       ; 継続: v1 は i 番目の要素
          ;; i+1 番目以降を作る
          (CPS-make-list-from generator num-items (+ i 1)
            (lambda (v2) ; 継続: v2 は i+1 番目以降のリスト
              (k (cons v1 v2)))))))) ;; cons して継続に返す
```

大きく変わるのはmake-list-from である。この関数は $(1)i$ 番目の要素を作る、$(2)i+1$ 番目以降のリストを作る、(3) 両者の結果をcons する、という 3 つの動作がある。そこで (1) の継続に (2) を、(2) の継続に (3) を行うようになる。

3.4 CPS による例外処理

上の CPS は例外処理に対応していない。そこでこれを拡張して例外処理に対応させる。対応のさせ方は以下のようになる:

- 途中でキャンセルが起こり得る関数は、(1) 正常時の継続と (2) キャンセル時の継続の 2 つを引数として受け取る
- キャンセル時には、「キャンセル時の継続」を呼び出すだけ。(正常時の継続を呼び出さなければ、「その後の計算」は捨てられる)
- 呼び出し側は、キャンセル時に行いたい動作を「キャンセル時の継続」として渡す

例えば、入力途中でキャンセルされたときに、「最初から入力し直す」には、input-values の中からmake-list を呼び出すときの「キャンセル時の継続」としてinput-values を最初から実行するようなものを与えればよい (Web ページcps-with-continuation.scm に定義あり。):

```
(define (CPS-input-values num-values k1)
  (let ((input-area (make <InputArea>)))
    (CPS-make-list
     (lambda (index k2 cancel)
       (CPS-input input-area (make-prompt index) k2 cancel))
     num-values
     k1 ;; 正常時の継続
     ;; キャンセル時の継続: 最初から入力し直す
     (lambda () (CPS-input-values num-values k1)))))
```

他の関数ではキャンセル時の動作は、呼び出し側で定めたものに従わせるので、渡された継続をそのまま渡してやるだけである。

```
(define (CPS-make-list-from generator num-items i k cancel)
  (if (= i num-items)
      (k '())                          ; 正常時
      (generator i                     ; i 番目の要素の作成
        (lambda (v1)                   ; その継続
          (CPS-make-list-from generator num-items (+ i 1)
            (lambda (v2) (k (cons v1 v2)))
            cancel))                   ; キャンセル時継続
        cancel)))                      ; キャンセル時継続

(define (CPS-make-list generator num-items k cancel)
  (CPS-make-list-from generator num-items 0 k cancel))
```

関数input は、メソッドinput の結果に従って正常時あるいはキャンセル時の継続を呼び出す:

```
(define (CPS-input input-area prompt k cancel)
  (let ((value-or-false (input input-area prompt)))
    (if value-or-false                 ; キャンセルか?
        (k value-or-false)             ; 正常時
        (cancel))))                    ; キャンセル時
```

このやり方の特徴をまとめる:

- 関数を呼び出した側は、途中でキャンセルされたどうかを判断する必要がない
- 値を「値」と「キャンセル」のように変換する必要がない
- プログラムを CPS にしなければならない

3.5 1 つ前に戻る

継続渡し形式になっていると、キャンセル時の動作を細く指定することが簡単になる。例えば、以下は i 番目の値の入力時にキャンセルされた場合に、$i-1$ 番目の入力に戻るようにしたものである。(Web ページstep-back.scm に定義あり。)

```
(define (CPS-make-list-from generator num-items i k cancel)
  (if (= i num-items)
      (k '())
      (generator i                          ; i番目を作る
        (lambda (v1)                        ; 正常時の継続
          ;; i+1番目以降を作る
          (CPS-make-list-from generator num-items (+ i 1)
            (lambda (v2) (k (cons v1 v2))) ; 正常時の継続
            (lambda ()                      ; キャンセル時の継続
              (CPS-make-list-from       ; i番目の入力をやり
               generator num-items i k cancel)))) ; 直す
        cancel)))                           ; キャンセル時の継続
```

ここでは i 番目以降の入力時に、次のようにキャンセル時継続を使い分けている:

- i 番目の入力中にキャンセルされたら、呼び出し側の決めたキャンセル動作に従う (つまり $i-1$ 番目の入力からやり直す)
- $i+1$ 番目以降の入力中にキャンセルされたら、i 番目の入力からやり直す

4 継続のとり出し

4.1 call-with-current-continuation

継続渡し形式の問題は、プログラムが複雑になりがちなことである。多くの Scheme 言語には、「現在の継続」をとり出す関数`call-with-current-continuation` がある。Scheme が`(call-with-current-continuation f)` という式を実行すると、その式の継続 c がとり出され、関数 `f` を引数 c とともに実行する。継続 c は、1 引数の関数のようにして扱える。つまり、`f` の中で$(c\ 3)$ のようにすると、`(call-with-current-continuation f)` の呼出元に返値 3 が渡って計算が続行される。

```
取り出された継続(cont)は関数のように使える
STk> (call-with-current-continuation
       (lambda (cont) (cont 1)))
1

何かの計算中に継続を呼び出した場合、途中の計算は捨てられる
STk> (call-with-current-continuation
       (lambda (cont)
         (* (cont 1) 2)))
1

継続を呼び出した場合、とり出された継続からの計算が続行される
STk> (/ (call-with-current-continuation
          (lambda (cont)
            (* (cont 1) 2)))
        2)
0.5
```

4.2 call-with-current-continuation によるキャンセル

`call-with-current-continuation` を使えばキャンセル時の継続を作ることができる (Web ページ`callcc.scm` に定義あり。):

```
(define (call-with-cancellation proc cancel-proc)
  (call-with-current-continuation
   (lambda (success-cont)
     (call-with-current-continuation
      (lambda (cancel-cont)
        (success-cont (proc cancel-cont))))
     (cancel-proc))))
```

このようにするとまず、外側の call-with-current-continuation によってsuccess-cont が関数全体の継続になる。次に内側のcall-with-current-continuation によって、cancel-cont がcancel-proc 以下を実行する継続になる。その上で、(success-cont (proc cancel-cont)) が実行される。

この関数は例えば以下のようにして使う:

```
(sqrt
 (call-with-cancellation
  (lambda (cancel)               ; 本体
    (let ((x ...input from user...))
      (if (< x 0) (cancel))
      x))
  (lambda ()                     ; 例外処理部
    (display "error") (newline)
    0)))
```

すると、cancel-cont は(sqrt (begin □ (display "error") (newline) 0)) のようなものになって本体部分が実行される。

これを使ってinput-values を書き直すと以下のようになる。(区別のため名前にcall/cc-を付けている。)

```
(define (call/cc-input-values num-values)
  (call-with-cancellation
   (lambda (cancel1)                        ; 本体部分
     (let ((input-area (make <InputArea>)))
       (make-list
        (lambda (index cancel2)
          (call/cc-input input-area (make-prompt index)
                         cancel2))
        num-values
        cancel1)))
   (lambda ()                                ; キャンセル処理部
     (call/cc-input-values num-values))))    ; 再実行
```

make-list やmake-list-from はキャンセル用の継続の受け渡しをするが、継続渡し形式のように関数定義を大幅に書き直す必要はない。

```
(define (call/cc-input input-area prompt cancel)
  (let ((value-or-false (input input-area prompt)))
    (or value-or-false                    ; 正常時
        (cancel))))                       ; キャンセル時

(define (make-list-from generator num-items i cancel)
  (if (= i num-items)
      '()
      (let ((v (generator i cancel)))
        (cons v
              (make-list-from generator num-items
                              (+ i 1) cancel)))))

(define (make-list generator num-items cancel)
  (make-list-from generator num-items 0 cancel))
```

参考: C++や Java のような命令型言語においては、例外ハンドラという概念がある。

- 例外ハンドラでは、例外を起こす際に例外処理をするハンドラを明示する必要はない (逆に継続では明示されているため、状況に応じて選択することができる)
- 「例外ハンドラ」自体は高階関数のように渡すことができない

練習 **12-1:** (1 つ前に戻る)

`call-with-cancellation` を使って、i 番目の入力時にキャンセルされたら $i-1$ 番目の入力に戻るように`call/cc-make-list-from` を定義せよ。 ∎