「プログラミング言語」
SICP 第4章
～超言語的抽象～
補足資料

五十嵐 淳
京都大学 工学部情報学科計算機科学コース
大学院情報学研究科通信情報システム専攻
工学部 10 号館 2 階 224 号室
e-mail: igarashi@kuis.kyoto-u.ac.jp

この配布資料では，教科書 4.3 節を理解するにあたって大切・必要と思われる概念である継続について，Scheme の `error` 関数や，例外処理機構の (メタサーキュラ) インタプリタ上への実装を通じて述べる．

4.2$\frac{1}{2}$.1 例外 (実行時エラー) と例外処理機構

実行時エラーもしくは**例外**(*exception*) は，計算を進めていく過程で起こる，なんらかの理由で計算を中断せざるをえない状況を示す．なんらかの理由の例としては，定義されていない変数の参照，関数でないものを適用，0 での除算などがある．以下は例外が発生する式の例である．

```
foo
(5 + 3)
(/ 2 0)
```

また，jakld を含む多くの Scheme 処理系では，例外を発生させるための `error` 関数が用意されている．この関数を呼ぶことによって，プログラムの実行を中断し (残りの処理を行わずに) プロンプトに戻ってくることができる．

```
(+ 3
  (begin (error "My error!") (display "This will be ignored.") 4))
```

しかし，例外が発生した途端にプログラム全体の実行が終わってしまうのは望ましくない場合もある．例外の発生によって中断した実行を再開する仕組みが**例外処理**(*exception handling*) 機構と呼ばれるものである．Scheme 処理系によっては例外処理機構が実装されているが，ここでは Scheme の

先祖である Lisp 処理系の多くに備わっている，catch/throw という機構について紹介し，これをメタサーキュラ・インタプリタ上で実装する．

例外処理の仕組みは，

- 例外の発生を示すための仕組み
- 例外の発生を検知して，例外によって中断された実行を回復させる仕組み

のふたつから成り立っており，この前者を担うのが特殊形式 throw，後者が特殊形式 catch である．まずは，例外の発生を示す特殊形式 throw から見ていこう．これは二引数の特殊形式で，

```
(throw 'a (+ 2 3))
```

のように使う．第一引数は例外の種類を表すための「タグ」で，どんなデータでもよいが，シンボルを使うことが多い．第二引数は例外的な状況に関する詳しい情報を表すためのデータである．

throw は単独で使った場合，error 関数とほぼ同じ働きをする．つまり，呼び出されたところでプログラムの実行が中断されて (残りの処理を行わずに) プロンプトに戻ってくる．

```
(define (my-list-ref l n)
  (cond ((< n 0) (throw 'negative-index n))
        ((zero? n) (car l))
        (else (my-list-ref (cdr l) (- n 1)))))

(my-list-ref '(1 2 3) 1)
⟹ 2
(my-list-ref '(1 2 3) -5)
⟹ ;; uncaught exception
```

catch は throw によって発生した例外によるプログラムの中断からの回復を行う．まずは，catch を使った具体例を示す．

```
(catch 'negative-index
  (my-list-ref '(1 2 3) 1))
⟹ 2
```

catch は，まず (my-list-ref ...) の評価を行う．結果が (例外の発生なく) 値になった場合はその値が catch 式全体の値となる．上の式では，(my-list-ref ...) の値は 2 なので，全体の値も 2 になる．一方，例外が発生した場合には，発生した例外のタグと catch 直後に書かれているタグが等しいかを判定する．等しい場合には throw された値が式全体の値になる．

```
(+ (catch 'negative-index
     (my-list-ref '(1 2 3) -5))
   1)
⟹ -4
```

この式の場合は (my-list-ref ...) の部分は値を返さず，throw によって，タグ 'negative-index のついた-5 が投げられる．投げられた例外のタグと catch に書かれたタグが等しいので，catch 式の値は -5 となり，全体の値は -4 になる．

タグが等しくない場合には，例外はここでは捕捉されずに，より外側にある catch が処理することになる．catch がない場合には単独で throw を使ったのと同じように uncaught exception であることを示してプロンプトに戻る．

```
(catch 'another-tag
  (my-list-ref '(1 2 3) -5))
⟹ ;; uncaught exception
```

最後に特殊形式 catch の一般的な形とその動作についてまとめる．

(catch *tag* exp_1 ... exp_n)

1. 式 *tag* を評価してタグとなる値を得る．この評価中に発生した例外はこの catch では捕捉されない (外側の catch があればそれが捕捉する).
2. 本体 exp_i を順に評価していく．評価が (通常) 終了したら最後の式 exp_n の値が全体の値となる．throw が実行された場合にはそのタグと上のタグを eq? を使って比較し，等しければ throw によって投げられた値を全体の値とする．等しくない場合は，(同じ) 例外が再発生する．(外側の catch があればそれが捕捉する．)

練習問題 1 整数リストの要素全ての積を返す関数 prod-list を定義せよ．リスト要素にひとつでも 0 が含まれている場合には prod-list の適用結果は常に 0 になるので，catch，throw を利用して，0 を発見したら残りの計算を行わずに中断して 0 を返すように定義せよ．

練習問題 2 リストとその要素を受け取り，要素が最初に現れるのがリストの何番目かを返す関数 pos-in-list を定義せよ．この時，与えられた要素がリストに現れない場合は -1 を返すようにせよ．(等しさの判定には eq? を用いよ．)

練習問題 3 以下の関数 change は，お金を「くずす」関数である．

```
(define (change coins amount)
  (if (zero? amount) '()
      (let ((c (car coins)))
        (if (> c amount) (change (cdr coins) amount)
          (cons c (change coins (- amount c)))))))
```

与えられた (降順にならんだ) 通貨のリスト coins と合計金額 total からコインのリストを返す．

```
(define us-coins '(25 10 5 1))      ;; American coins
(define gb-coins '(50 20 10 5 2 1)) ;; British coins
(change gb_coins 43)
(change us_coins 43)
```

しかし，この定義は先頭にあるコインをできる限り使おうとするため，可能なコインの組合せがあるときにでも失敗してしまうことがある．

```
(change '(5 2) 16)
```

これを，例外処理を用いて解がひとつでも存在する場合には答えが得られるようにすることを考える．以下の 2 個所の ... を埋めて関数定義を完成させよ．

```
(define (change coins amount)
  (cond ((zero? amount) '())
        ((null? coins) ...)
        (else
          (let ((c (car coins)))
            (if (> c amount) (change (cdr coins) amount)
                (or (catch 'fail
                      (cons c (change coins (- amount c))))
                    ...)))))
```

4.2$\frac{1}{2}$.2　継続

さて，**継続**(*continuation*) 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，残りの作業のことである．TODO リスト (これからやることのリスト) といってもよいかもしれない．この TODO リストは，作業を完了することで縮んだり，作業をさらに細かい作業列に分割することで伸びたりもするものである．

例えば，(+ (* e_1 e_2) e_3) の評価プロセスを考える．eval の最初の場合分けが終わった時点での継続は

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. (apply により) v_1 と v_2 の和を求める．

であると考えられる．(厳密には,「和を求める」のかどうかの判断は apply の中で行われる．)

もう少し評価を進め，(* e_1 e_2) に関する eval の場合分けが終わった時点での継続は,「(* e_1 e_2) の評価」という作業を細分化した

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. (apply により) v_{11} と v_{12} の積を求める (値を v_1 とする).
4. e_3 の評価をする (値を v_2 とする).
5. (apply により) v_1 と v_2 の和を求める．

と考えられる．さらに評価を進め，e_1 の値が求まった瞬間の継続を考えると，これは，ひとつめの作業が完了したので，

1. e_2 の評価をする (値を v_{12} とする).
2. (`apply` により) e_1 の値 v_{11} と v_{12} の積を求める (値を v_1 とする).
3. e_3 の評価をする (値を v_2 とする).
4. (`apply` により) v_1 と v_2 の和を求める．

という短かいものになる．このように，評価を進めるにつれて継続が変化してゆく．

また，式の形の上では外側に書かれている操作—この例では `+` の適用—が継続の中ではより後ろに現れることになることに注意してもらいたい．

例外と継続　さて，ここで例外を発生させる関数 `error` の動作について継続を使って考えてみる．`error` の動作は「計算を中断してプロンプトに戻る」ことであるが，これは継続を使って言い換えると「`error` が呼ばれた時点での継続を全て捨てる」ことと考えることができる．

例えば，`(+ (error` e`) (* 3 4))` の評価を考えると，e の評価が終わった時点での継続は，前の例と同じようなプロセスを経て，

1. `error` を e の値で呼び出す (その値を v_1 とする).
2. `(* 3 4)` を評価する (値を v_2 とする).
3. (`apply` により) v_1 と v_2 の和を求める．

となっていると考えられるが，呼び出された `error` は，継続を捨てることで，即座に計算を終了するのである．

`catch` と `throw` の動作も同様に継続の操作として捉えることができる．まず，`catch` は，本体の評価に入る時に，継続に「`catch` の本体の終わり」を示す印をつけておく．この際にタグが何であったかもいっしょに記録しておく．この印に，本体の計算が終わった後の次の作業として出会った場合には，何もせず次に置かれている作業に移る．この印は，本体の計算中に `throw` が実行される際により大きな意味を持つ．`throw` は，`error` と同様に継続を捨てるのだが，全部ではなく，一番最初の `catch` による印のところまでを捨てる．そして，一緒に記録されているタグと `throw` されたタグを比較し，等しいなら継続に書かれている次の動作に移り，等しくないなら，`catch` による印のところまでの継続を捨て，タグを比較し，という動作を繰り返す．

このように例外処理機構を実装するには，インタプリタ内部で継続がデータとして扱えれば十分である．このような，継続がデータ化され操作の対象となっているようなインタプリタを**継続渡しインタプリタ**(*continuation-passing interpreter*) と呼ぶ．

以下では，まず，例外機構を持たない (遅延評価もない) Scheme の継続渡しインタプリタの実装について述べてから，`catch`/`throw` の実装を示す．

4.2$\frac{1}{2}$.3　継続渡しインタプリタ

4.2$\frac{1}{2}$.3.1　継続渡しインタプリタの構造

4.1 節のインタプリタで主要な役割を果たしていたのは

- 式の形に応じて場合分けを行ない，それに応じた適切な動作を割り当てる eval 関数
- 関数適用の動作を記述した apply 関数

であり，これらが相互に呼び合うことで式の評価を行っていた．

継続渡しインタプリタでは，まず eval 関数が，式，環境に加えて，(与えられた式の評価が終わったら行う残りの計算を表現する) 継続を引数とする点が大きな違いである．そして，もうひとつ apply-cont という関数が主要な役割を果たすものとして加わる．この関数は，継続と値を引数として，継続が表現する「残りの計算」を行うものである．eval と apply-cont も相互に呼び合う構造になっており，

- eval (もしくは，そこから呼ばれる eval-XXX とい補助関数) は，値が得られた (つまり，今与えられた仕事が終わった) 際に apply-cont を呼ぶことで残りの計算を起こす．
- apply-cont は，継続の「TODO リスト」を見て，式の評価を行う必要があれば，eval を呼ぶ

という関係になっている．(後で見るように，実際には eval と apply と apply-cont が三つ巴になって呼び合う構造になっている．)

eval の定義　図 1 に eval の定義を示す．まず，上で述べたように 3 つめの引数として継続を表わす cont が加わっている．次に，再帰的に eval を呼び出すことなく値が得られる場合—すなわち，定数，変数，引用，ラムダの場合—には，apply-cont に得られた値を与えて残りの計算を開始している．

次に，4.1 節のインタプリタにおいて，再帰的な eval の呼び出しを行っていた場合の処理が，ここではどうなっているかを見てみよう．図 2 に，eval-if などの補助関数群を示す．(比較のためにコメントとして eval-if の元の定義を示した．) eval-if で行う計算は，

1. 条件部 (if-predicate) の式を評価する
2. 値が true? を満たすならば if-consequent 部の式を，そうでないならば if-alternative 部の式を評価し，その値を if 全体の値とする

というものであった．継続渡しインタプリタのポイントは，

最初の作業以外を全て与えられた継続に追加して (追加した新しい継続を作って)，eval を再帰呼び出しする

というところにある．新しい eval-if の定義で，make-testc が新しい継続を作る関数であり，これを使って，条件部 (if-predicate) の式を評価するべく eval の再帰呼び出しを行っている．条件部の値が得られた暁には，その値を使って残りの計算である「条件判定とそれに応じた式の評価」が行われることになる (その部分の実際の処理は apply-cont が担当する．)

```
(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)))
   ((assignment? exp) (eval-assignment exp env cont))
   ((definition? exp) (eval-definition exp env cont))
   ((if? exp) (eval-if exp env cont))
   ((lambda? exp)
    (apply-cont cont
     (make-procedure (lambda-parameters exp)
                     (lambda-body exp)
                     env)))
   ((begin? exp)
    (eval-sequence (begin-actions exp) env cont))
   ((cond? exp) (eval (cond->if exp) env cont))
   ((application? exp)
    (eval (operator exp) env
          (make-operandsc (operands exp) env cont)))
   (else
    (error "Unknown expression type -- EVAL" exp))))
```

図 1: 継続渡しインタプリタ (1)

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

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

(define (eval-sequence exps env cont)
  (cond ((last-exp? exps) (eval (first-exp exps) env cont))
        (else (eval (first-exp exps) env
                    (make-beginc (rest-exps exps) env cont)))))

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

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

図 2: 継続渡しインタプリタ (2)

以降では，make-XXXc (c は continuation の c) を継続を作る関数の名前として使う．継続を作る関数は，残りの計算を行うために必要となる情報—make-testc であれば，if-consequent 部の式と if-alternative 部の式，それにそれらを評価する際の環境—を引数とする．もちろん，さらに if の残りの計算も必要となるので，最初に与えられた継続も覚えておかなければいけない．(継続が「TODO リスト」であるとすると，make-XXXc はリストに要素を追加する cons 相当である．)

eval-if と同様，eval-sequence なども，残りの作業をひとまず継続に追加して eval を再帰呼び出しするだけで終わっている．また，eval の application? の場合も，関数部を評価するための eval の再帰呼び出しに与える継続として，make-operandc を使って「引数部を評価する」という計算を追加した継続を与えている．

ここまでに現れた継続を作る関数 (の名前の XXXc 部分) とそれが表す計算の内容を整理してみると以下のようになる．

testc	条件判定をして，適当な式の評価を行う
beginc	begin の残りの式の評価を行う
assignc	変数への代入を行い，'ok を返す
definec	変数定義を行い，'ok を返す
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

apply-cont の定義　次に，apply-cont の定義を見てみよう (図 3)．apply-cont は継続と，直前の計算で得られた値を引数とする関数で，継続の種類についての場合分けを行い，その種類が表す計算の内容を実行するものとなっている．

最初の 5 つの場合が，既に見た継続の種類についての処理となっている．(make-XXXc に対応して XXXc? という関数で継続の種類を判定している．) 例えば，testc? の場合には，与えられた val が真かどうかを判定し，それに応じた式を評価している． ここの処理が，実質的に 4.1 節の eval-if の後半部分と同じである，というのがポイントである．

testc-true, testc-false, testc-env, testc-cont は継続が保持している条件部が真だった場合に評価すべき式，条件部が偽だった場合に評価すべき式，それらの式を評価する際に使われるべき環境，if のさらにその残りの計算を表す継続，をそれぞれ取り出す関数である．

同様に他の場合も，4.1 節のインタプリタでは eval-define などに書かれていた処理の後半部分と同様なことが書かれている．ただし，define，set! のような 'ok 値を返す場合は，新しい eval の中の定数式の処理と同様に，apply-cont を呼び出すことで，継続の処理に入っていく．

operandsc? の場合，継続が表している作業が

1. 引数リストを順次評価し，値のリストを作る
2. その引数の値のリストと val (関数を表しているはずである) で apply を呼び出す

という二段階になっているので，それを分割して，後半に相当する継続を make-applyc で作り，前半に相当する計算を行う list-of-values (図 4) を呼び出している．make-applyc で作られた継続についての処理は単に apply を呼び出すだけである．

list-of-values で行うべき計算も

1. 最初の引数の評価
2. 残りの引数の評価
3. それぞれの結果を cons で繋ぐ

というように分割され，残りの引数の評価以降の計算の継続化は make-restopsc で表されている．ここで作られた継続についての処理は apply-cont の

```
(define (apply-cont cont val)
  (cond ...
        ((restopsc? cont)
         (list-of-values (restopsc-rest cont) (restopsc-env cont)
                         (make-consc val (restopsc-cont cont))))
        ...
))
```

に書かれており，残りの引数の評価を list-of-values でするとともに,「それぞれの結果を cons で繋ぐ」を表す継続を make-consc で作っている．この際，直前の計算の結果として得られたばかりの「最初の引数の評価結果」(val) を継続に格納しておく必要がある．

この継続についての処理は，直前の計算で得られたばかりの「残りの引数の評価結果リスト」(val) と，継続に格納されていた「最初の引数の評価結果」(consc-val) を cons し，残りの計算を呼び出すことになる．

```
(define (apply-cont cont val)
  (cond ...
        ((consc? cont)
         (apply-cont (consc-cont cont)
                     (cons (consc-val cont) val)))
        ...
))
```

apply の定義: apply の定義は，実はほとんど以前と変わらない (図 5)．主な変更点は，

- 関数適用の結果得られる値を渡す継続を引数とすること
- プリミティブの場合はプリミティブを適用する apply-primitive-procedure に，ユーザ定義関数の場合は本体を評価する eval-sequence に，その引数を渡すこと

くらいである．プリミティブを適用するとその結果，値が得られるので apply-cont を使って，継続をその値に適用している．

REPL の実装 REPL[1]については，実行された式を eval に渡す際，ここで評価がおしまい，ということを示す継続を make-haltc 関数で作る．

この継続は単に継続に与えられた値を返せばよい．以下はその部分に対応する apply-cont の断片である．

```
(define (apply-cont cont val)
  (cond ...
        ((haltc? cont) val)
))
```

継続のデータ表現 継続に関する関数については，図 6, 図 7 に示す．また各 make-XXXc 関数で構成される継続の役割について，まとめ直す．

testc	条件判定をして，適当な式の評価を行う
assignc	変数への代入を行い，'ok を返す
definec	変数定義を行い，'ok を返す
beginc	begin の残りの式の評価を行う
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
applyc	関数を適用する (operandsc が表す計算の最後の部分でもある)
restopsc	残りの引数を評価し，最初の引数の値と cons で繋ぐ
consc	(restopsc の表す計算の一部で) 最初の引数の値と残りの引数の値を cons する
haltc	一連の評価の最後 (継続リストの末尾) を表し，値の出力を行う

[1] read-eval-print-loop: 第 4 章一回目のスライド参照

```
(define (apply-cont cont val)
  (cond ((testc? cont)
         (if (true? val)
             (eval (testc-true cont) (testc-env cont) (testc-cont cont))
             (eval (testc-false cont) (testc-env cont) (testc-cont cont))))
        ((assignc? cont)
         (set-variable-value! (assignc-var cont) val (assignc-env cont))
         (apply-cont (assignc-cont cont) 'ok))
        ((definec? cont)
         (define-variable! (definec-var cont) val (definec-env cont))
         (apply-cont (definec-cont cont) 'ok))
        ((beginc? cont)
         (eval-sequence
          (beginc-rest-exps cont)
          (beginc-env cont)
          (beginc-cont cont)))
        ((operandsc? cont)
         (list-of-values (operandsc-exps cont)
                         (operandsc-env cont)
                         (make-applyc val (operandsc-cont cont))))
        ((applyc? cont)
         (my-apply (applyc-proc cont) val (applyc-cont cont)))
        ((restopsc? cont)
         (list-of-values (restopsc-rest cont) (restopsc-env cont)
                         (make-consc val (restopsc-cont cont))))
        ((consc? cont)
         (apply-cont (consc-cont cont)
                     (cons (consc-val cont) val)))
        ((haltc? cont) val)
        (else (error "Unknown continuation type -- APPLY-CONT" cont))
))
```

図 3: 継続渡しインタプリタ (3)

```
(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))))
```

図 4: 継続渡しインタプリタ (4)

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

(define (apply-primitive-procedure proc args cont)
  (apply-cont cont (apply (primitive-implementation proc) args)))
```

図 5: 継続渡しインタプリタ (5)

```
(define (make-testc true-exp false-exp env cont)
  (list 'testc true-exp false-exp env cont))
(define (testc? cont) (tagged-list? cont 'testc))
(define (testc-true cont) (cadr cont))
(define (testc-false cont) (caddr cont))
(define (testc-env cont) (cadddr cont))
(define (testc-cont cont) (car (cddddr cont)))

(define (make-beginc exps env cont)
  (list 'beginc exps env cont))
(define (beginc? cont) (tagged-list? cont 'beginc))
(define (beginc-rest-exps cont) (cadr cont))
(define (beginc-env cont) (caddr cont))
(define (beginc-cont cont) (cadddr cont))

(define (make-assignc var env cont) (list 'assignc var env cont))
(define (assignc? cont) (tagged-list? cont 'assignc))
(define (assignc-var cont) (cadr cont))
(define (assignc-env cont) (caddr cont))
(define (assignc-cont cont) (cadddr cont))

(define (make-definec var env cont) (list 'definec var env cont))
(define (definec? cont) (tagged-list? cont 'definec))
(define (definec-var cont) (cadr cont))
(define (definec-env cont) (caddr cont))
(define (definec-cont cont) (cadddr cont))

(define (make-operandsc exps env cont)
  (list 'operandsc exps env cont))
(define (operandsc? cont) (tagged-list? cont 'operandsc))
(define (operandsc-exps cont) (cadr cont))
(define (operandsc-env cont) (caddr cont))
(define (operandsc-cont cont) (cadddr cont))
```

図 6: 継続渡しインタプリタ (6)—継続の表現 (1)

```
(define (make-applyc proc cont)
  (list 'applyc proc cont))
(define (applyc? cont) (tagged-list? cont 'applyc))
(define (applyc-proc cont) (cadr cont))
(define (applyc-cont cont) (caddr cont))

(define (make-restopsc exps env cont)
  (list 'restopsc exps env cont))
(define (restopsc? cont) (tagged-list? cont 'restopsc))
(define (restopsc-rest cont) (cadr cont))
(define (restopsc-env cont) (caddr cont))
(define (restopsc-cont cont) (cadddr cont))

(define (make-consc val cont)
  (list 'consc val cont))
(define (consc? cont) (tagged-list? cont 'consc))
(define (consc-val cont) (cadr cont))
(define (consc-cont cont) (caddr cont))

(define (make-haltc) 'haltc)
(define (haltc? cont) (eq? cont 'haltc))
```

図 7: 継続渡しインタプリタ (7)—継続の表現 (1)

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

(define (catch? exp) (tagged-list? exp 'catch))
(define (catch-tag exp) (cadr exp))
(define (catch-body exp) (cddr exp))

(define (throw? exp) (tagged-list? exp 'throw))
(define (throw-tag exp) (cadr exp))
(define (throw-body exp) (caddr exp))
```

図 8: catch/throw の実装 (1): eval と式の表現

4.2$\frac{1}{2}$.4 catch/throw の実装

ここまでが理解できれば catch と throw の実装は最初の説明とあわせて考えればそれほど難しくはない．まず，catch については，

1. タグ部の評価
2. 継続に「印」をつけて本体の評価

にわかれているので，後半の計算を表現する継続が必要となる．同様に throw についても，

1. タグ部の評価
2. (投げられる) 式の評価
3. 例外の発生 (継続中の「印」の探索)

にわかれているので，残り 2 ステップの計算を表現する継続が必要となる．これらをそれぞれ make-cabodyc, make-thbodyc という関数を呼ぶことで作る．よって，eval の変更は図 8 のようになる．(図には catch, throw 両特殊形式のデータ表現に関する関数も示している．)

次に，apply-cont の変更を図 9 に示す．

catch 式の本体は式の列なので，eval-sequence を使って評価を行う．この時，継続に catch に「印」をつける．「印」をつけた継続を作るのが make-catchc である．この「印」には，直前で評価された値，すなわち catch 式のタグ情報を埋めこんでおく．また，catchc? の節 (の本体) が実行され

```
(define (apply-cont cont val)
  (cond ...
    ((cabodyc? cont)
     (eval-sequence (cabodyc-body cont) (cabodyc-env cont)
                    (make-catchc val (cabodyc-cont cont))))
    ((catchc? cont)
     (apply-cont (catchc-cont cont) val))
    ((thbodyc? cont)
     (eval (thbodyc-body cont) (thbodyc-env cont)
           (make-throwc val (thbodyc-cont cont))))
    ((throwc? cont)
     (let ((stripped-cont
             (first-matching-catch (throwc-tag cont) (throwc-cont cont))))
       (if stripped-cont
           (apply-cont stripped-cont val)
           (list 'uncaught (throwc-tag cont) val))))
    ((haltc? cont) (list 'normal val))
    ...
))
```

図 9: catch/throw の実装 (2): apply-cont

るのは，catch 式本体で throw がなく評価が終わった場合である．この時は単に得られた値をその外側の継続に与えればよい．

thbodyc? の節では，throw の，(投げられる値となる) ふたつめの引数の計算を行う．make-throwc が表現する継続の動作は，例外を発生させることであり，その動作は，この apply-cont の最後の節に書かれている．そこでは，first-matching-catch という関数を使って，継続の中からこの throw を捕捉する catch の印を見つけ，その継続を投げられた値に適用する．見つからない場合，first-matching-catch は false を返すので，その場合は，throw されたタグと値に 'uncaught をつけて (例外が catch されなかったという) 評価結果とする[2]．図 10 が，継続を探索する関数である．基本的には，「印」となる継続 (catchc? をみたす) を見つけるまで，継続のリストをたどっていくという構造である．catchc? をみたす継続が見つかると，throw 側のタグと catch 側のタグ (catchc-tag で取得できる) の比較を行い，一致していたら，catch のすぐ外側の継続を返す．

図 11 に catch と throw に関係する継続のデータ表現を規定する関数群を，図 12 に変更された driver-loop 関数の定義を示しておく．

練習問題 4 ここで導入した catch 式は，タグさえ合えば throw された値を**そのまま**返してしまうので，本体の評価が正常に終了した時と，何かが throw された時で別の処理をしたい場合にはやや不便である．

[2] と，同時に，評価が正常終了した場合 (apply-cont の haltc? の節) は，式の値に'normal をつけて評価結果とする．それに応じて，driver-loop の評価結果の表示部分も修正が必要になる．(図 12 参照)

```
(define (first-matching-catch thrown-tag cont)
  (define (loop cont)
    (cond ((haltc? cont) false)
          ((testc? cont) (loop (testc-cont cont)))
          ((assignc? cont) (loop (assignc-cont cont)))
          ((definec? cont) (loop (definec-cont cont)))
          ((beginc? cont) (loop (beginc-cont cont)))
          ((operandsc? cont) (loop (operandsc-cont cont)))
          ((applyc? cont) (loop (applyc-cont cont)))
          ((restopsc? cont) (loop (restopsc-cont cont)))
          ((consc? cont) (loop (consc-cont cont)))
          ((cabodyc? cont) (loop (cabodyc-cont cont)))
          ((catchc? cont)
           (if (eq? thrown-tag (catchc-tag cont))
               (catchc-cont cont)
               (loop (catchc-cont cont))))
          ((thbodyc? cont) (loop (thbodyc-cont cont)))
          ((throwc? cont) (loop (throwc-cont cont)))
          (else (error "Unknown continuation type -- FIRST-MATCHING-CATCH" cont))))
  (loop cont))
```

図 10: catch/throw の実装 (3): 継続探索処理

```
(define (make-cabodyc body env cont) (list 'cabodyc body env cont))
(define (cabodyc? cont) (tagged-list? cont 'cabodyc))
(define (cabodyc-body cont) (cadr cont))
(define (cabodyc-env cont) (caddr cont))
(define (cabodyc-cont cont) (cadddr cont))

(define (make-catchc tag cont) (list 'catchc tag cont))
(define (catchc? cont) (tagged-list? cont 'catchc))
(define (catchc-tag cont) (cadr cont))
(define (catchc-cont cont) (caddr cont))

(define (make-thbodyc body env cont) (list 'thbodyc body env cont))
(define (thbodyc? cont) (tagged-list? cont 'thbodyc))
(define (thbodyc-body cont) (cadr cont))
(define (thbodyc-env cont) (caddr cont))
(define (thbodyc-cont cont) (cadddr cont))

(define (make-throwc tag cont) (list 'throwc tag cont))
(define (throwc? cont) (tagged-list? cont 'throwc))
(define (throwc-tag cont) (cadr cont))
(define (throwc-cont cont) (caddr cont))
```

図 11: catch/throw の実装 (4)—継続の表現

```
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let* ((input (read))
         (output (eval input the-global-environment (make-haltc))))
    (cond ((tagged-list? output 'normal)
           (announce-output output-prompt)
           (user-print (cadr output)))
          ((tagged-list? output 'uncaught)
           (display ";;; Uncaught exception: ")
           (display (cadr output))
           (display " ")
           (user-print (caddr output))))
    (driver-loop)))
```

図 12: catch/throw の実装 (5)—`driver-loop`

catch 式を，例外が発生した場合には例外ハンドラと呼ばれる関数が呼ばれるように変更せよ．具体的には，catch 式を

(catch *tag handler* exp_1 ... exp_n)

という形式に拡張する．そして，本体中で throw が実行され，タグの比較で等しかった場合には *handler* を throw によって投げられた値に適用して得た値を全体の値とするように評価器を変更せよ．

そして，上の練習問題の change を新しい catch を使って，以下のように書き換えて動作を確かめよ．

```
(define (change coins amount)
  (cond ((zero? amount) '())
        ((null? coins) ...)
        (else
          (let ((c (car coins)))
            (if (> c amount) (change (cdr coins) amount)
                (catch 'fail
                   (lambda (x) ...)
                   (cons c (change coins (- amount c)))))))))
```