「プログラミング言語」
SICP 第4章
～超言語的抽象～
その4

五十嵐 淳

igarashi@kuis.kyoto-u.ac.jp

京都大学

July 10, 2013

今日(と来週)のメニュー

4.3 **への準備運動**:

- **前回・前々回の宿題**
- 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**

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

- 例外: これ以上計算が続けられない，計算を中断せざるをえない状況
  - 定義されていない変数の参照
  - ゼロでの除算
  - 関数でないものの適用
  - ...
- `error` 関数—プログラマによる意図的な実行の中断

例外発生による実行の中断

例外が発生しなかったら実行されていたはずの処理は実行されない

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

なら，`display`**や足し算は実行されない**

例外処理機構

- 「例外発生，即プログラムの実行終了」では困る場合も多い
  - 「読み書きしようとしたファイルが存在しない」という例外

$\Rightarrow$ ファイル名を再入力させたい

- 例外処理機構 = 中断した実行を**再開**させて，正常時の処理の流れに復帰させる仕組み
- 現代的なプログラミング言語には必須の機構
  - 言語によって微妙な差異はあるが，基本アイデアはみな同じ

catch/throw 機構

Scheme の先祖の Lisp (の多く) に備わっている:

- throw: 例外発生のための特殊形式

  (throw ⟨タグ式⟩ ⟨式⟩)

  - ⟨タグ式⟩は (ふつう) 例外の種類を表すシンボル
  - ⟨式⟩は例外発生と一緒に伝えるべき情報
  - 意味: ここで実行を中断して，最も**近い** catch に ⟨式⟩を**投げる**
- catch: 例外による中断からの復帰のための特殊形式

  (catch ⟨タグ式⟩ ⟨式⟩ ... ⟨式⟩)

  - 意味: ⟨式⟩ ... ⟨式⟩を順に実行していく．実行中に発生した (⟨タグ式⟩の類いの) 例外を**捕捉する**

catch/throw 入門(1/4)

throw は単独で使えば error と同じく単なる実行中断(プロンプトに戻る):

```
(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/throw 入門(2/4)

catch は throw が実行されなければ，何もないのと同じ:

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

catch/throw 入門(3/4)

throw された例外のタグと，catch しようとしているタグが等しいなら，catch 式の値は throw された値になる

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

catch/throw 入門(4/4)

タグが等しくないなら，その `catch` は無視される

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

練習問題 プレ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．

解答例:

```
(define (pos-in-list l elm)
   (cond
     ((eq? (car l) elm) 1)
     (else (+ 1 (pos-in-list (cdr l) elm)))))
```

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．

catch/throw **を使わない解答例**:

```
(define (pos-in-list l elm)
  (cond
    ((null? l) -1)
    ((eq? (car l) elm) 1)
    (else
      (let ((n (pos-in-list (cdr l) elm)))
        (if (= n -1) -1 (+ 1 n))))))
```

catch/throw を使った解答例:

```
(define (pos-in-list l elm)
  (define (pos-in-list-aux l elm)
    (cond
      ((null? l) (throw 'not-found -1))
      ((eq? (car l) elm) 1)
      (else
        (+ 1 (pos-in-list-aux (cdr l) elm)))))
  (catch 'not-found (pos-in-list-aux l elm)))
```

- **要素が見つからなかった場合の処理の分離**
- i.e, **「ふつうの場合」の処理はほぼそのまま**

4.2$\frac{1}{2}$.2: 継続 (continuation)

「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

TODO リストの管理・更新

今朝の TODO リスト

7/10 1限: 「プログラミング言語」の講義

7/10帰り: 牛乳を買う

7/11: 次回「プログラミング言語」の準備

7/12: ○○会議

TODO リストの管理・更新

明日正午の TODO リスト

- ~~7/10 1限: 「プログラミング言語」の講義~~
- ~~7/10帰り: 牛乳を買う~~
- 7/11: ~~次回「プログラミング言語」の準備~~
  - 宿題を考える
  - スライドを作る
- 7/12: ○○会議

TODO は詳細化される，終われば消える

評価における TODO リスト

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

評価における TODO リスト

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

式の評価における TODO リストの特徴

- 式の形についての場合分けによって詳細化される
- 内側の式に関する作業ほど前の方に来る
- 今評価している式の外側にどんな式があったかは TODO リストを見ればわかる

例外と継続

例外処理　　継続に対する操作

- `error` **の実行**　　TODO **リストを捨てる**
  - jakld **では** TODO **リスト (の一部の情報) が** backtrace **として表示される**
- `throw` **の実行**　　**最も近い** `catch` **までの** TODO **リストを捨てる**

catch の実行

例: `(+ 1 (catch 'a e))` **の実行**

`catch` を実行する直前の継続

1. `catch` **式を評価する**
2. **その値に** `1` **を足す**

e 評価直前の継続

1. ~~`catch` 式を評価する~~
   1. e **を評価し，その値を** v **とする**
   2. `catch` **式全体の値を** v **とする**
      - `catch 'a` **の範囲の終了** **を示す**
2. **その値に** `1` **を足す**

e 中の (throw 'a e0) 評価直前の継続

1. e0 を評価する
2. その値を throw する
3. ⋮
4. catch 式全体の値を v とする
   - catch 'a の範囲の終了 を示す
5. その値に 1 を足す

e0 評価直後の継続

1. ~~*e0* を評価する~~
2. その値 (仮に `0` だったとする) を `throw` する
3. ⋮
4. `catch` 式全体の値を v とする
   - `catch 'a` の範囲の終了 を示す
5. その値に `1` を足す

その後の計算:

e0 評価直後の継続

1. ~~*e0* を評価する~~
2. その値 (仮に `0` だったとする) を `throw` する
3. ⋮
4. `catch` 式全体の値を v とする
   - `catch 'a` の範囲の終了 を示す
5. その値に `1` を足す

その後の計算:

- 「`catch 'a` の範囲の終了」までの継続を捨てる

e0 評価直後の継続

1. ~~*e0* を評価する~~
2. その値 (仮に 0 だったとする) を throw する
3. ⋮
4. catch 式全体の値を v とする
   - catch 'a の範囲の終了 を示す
5. その値に 1 を足す

その後の計算:

- 「catch 'a の範囲の終了」までの継続を捨てる
- catch 式全体の値 v を throw された 0 として計算を続ける

e0 評価直後の継続

1. ~~*e0* を評価する~~
2. その値 (仮に 0 だったとする) を throw する
3. ⋮
4. catch 式全体の値を v とする
   - catch 'a の範囲の終了 を示す
5. その値に 1 を足す

その後の計算:

- 「catch 'a の範囲の終了」までの継続を捨てる
- catch 式全体の値 v を throw された 0 として計算を続ける
- 最終結果は 1 (0 に 1 を足した結果)

例外処理の実装に向けて

例外処理をインタプリタに実装するには，

継続をデータとして評価器で操作

できれば十分!

$\Rightarrow$

継続渡しインタプリタ(continuation-passing interpreter)

宿題： 7/11 午前8時 締切

- 配布資料の練習問題 1 と 3
- 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ら、その人の名前を明記
- web は出典を明記 (「同じ」回答は減点)