Common Lisp における例外処理 —— Condition System の活用

株式会社数理システム 知識工学部
工藤 千加子
黒田 寿男

2006 年 8 月 3 日

$Id: chika-report.tex,v 1.5 2007/07/14 12:19:29 kuroda Exp $

目 次

1 はじめに 2

2 Condition システム 3
 2.1 関数仕様の重要性 3

3 Condition システムの使い方 4
 3.1 Condition 4
  3.1.1 Condition の定義と生成 (`define-condition`, `make-condition`) 4
 3.2 Signaling 5
  3.2.1 `error` 関数 6
  3.2.2 `cerror` 関数 6
  3.2.3 `signal` 関数 8
  3.2.4 `warn` 関数 8
  3.2.5 デバッガの強制起動 10
 3.3 Handling 15
  3.3.1 `handler-bind` マクロ 16
  3.3.2 `handler-case` マクロ 17
  3.3.3 `ignore-errors` マクロ 19
  3.3.4 default handler を使う 21
 3.4 Restart の設定 22
  3.4.1 `restart-case` マクロ 23
  3.4.2 `restart-bind` マクロ 27
 3.5 Restart の実行 31
  3.5.1 `invoke-restart` 31
  3.5.2 `invoke-restart-interactively` 32
  3.5.3 今現在 active な restart の取得 (`compute-restarts`, `find-restart`) 33

3.5.4 定義済み restart 関数 (abort, muffle-warning, continue, use-value, store-value) 34
3.6 Assertion 35
3.6.1 assert 35
3.6.2 check-type 36
3.6.3 ccase 37
3.6.4 ecase 37
3.6.5 ctypecase 38
3.6.6 etypecase 39

4 condition システムの活用 **40**
4.1 設計におけるコツ 40
4.1.1 condition の分類の重要性 40
4.1.2 ハンドリング場所の重要性 41
4.2 アプリケーションをデーモンプロセスとして動かすときには 44
4.2.1 スレッドが停止しないか注意 44
4.2.2 zoom の重要性 44
4.3 Thread 間のシグナリング 46

5 最後に **49**

1 はじめに

Common Lisp の関数を記述する際に「通常の状況」と「例外的状況」を区別することはときとしてとても有用です。

例えば、我々が関数 + を語るとき、通常の状況下においては引数として数値が渡ってくることが前提となりますし、read-char は、与えられた stream から次の character を取り出すものとして理解します。

一方、+ に数値以外の引数が渡されるという例外的状況が考えられます。Common Lisp の + は、数値以外の引数が渡されると type-error を通知します。例外的状況というのは必ずしも error や warning であるとは限りません。read-char が読みにいった stream に、既に読むべき character が無かった、即ち EOF という例外的状況に遭遇したときに、プログラムがその後正常に動作を継続するというのは多くの場合可能です。もちろん、言語処理アプリケーションにおけるパージングの途中での EOF など error として処理すべきケースもありますので、read-char が EOF を検知した場合には end-of-file を呼び出し元に通知する、というプロトコルを決めておくことは重要になります。

プログラムが例外に遭遇した場合にとる方法は、識別コードを返す、代わりの値を返す、変数の値を設定しなおしてリトライ、関数を定義しなおしてリトライ、制御の流れを変える、デバッガを呼び出す、などコンテキストに応じて様々です。

Common Lisp には、こういった例外処理の機能が非常に豊富に備わっており、例外を通知し処理する構文に加え、通知した箇所以降の処理を継続させるための構文があります。例えば、warning を通知し、それを受け取った上位関数側で何らかの処理を行なった後、その warning を通知した箇所以降の処理を継続させることができます。

もちろん、他のプログラミング言語においても多くの場合、例外を通知・処理する構文は与えられていますし、大域ジャンプ文などを利用して継続処理を実現することはできるでしょう。しかし、ここで重要なことは、Common Lisp では、例外を通知し、受け取り、必要に応じて処理を継続する機能が「ANSI 標準の言語仕様として定められている」ということです。これにより、プログラマは自分勝手なコーディングをせずに共通のヴォキャブラリにおいて例外処理を実現できるため、非常に見通しの良いコードが書ける上に、ANSI Common Lisp 準拠の Lisp インプリメンテーション上では、その部分がソースコードの変更なしにポートできることが保証されます。これは、長年にわたって使われるアプリケーションの開発には重要なことです。

このレポートでは、 Common Lisp における例外処理機能の基本的な使い方と、実際のアプリケーション開発における活用方法を、サンプルコードを中心にレポートします[1]。

[1] 3 章で説明する基本的な使い方に関しては、既に Common Lisp を知っている人には退屈な内容が多くありますが、Common Lisp の勉強を初めて間もない方の理解の助けになればと思いレポートにしました。また、章立ての都合上、サンプルの中には後の章で説明する機能を使ったものも多くありますが、ご了承ください。

2 Condition システム

「関数処理中に何らかの通知すべき状態が発生し、それを通知する。通知を受けて何らかの処理を行なう。そして処理を継続させる。」といった一連の機能を、Common Lisp では『Condition システム』と呼びます。それぞれ次のように名前付けられています。

Condition	通知する状態
Signaling	Condition を通知する機能
Handling	Condition を受け取る機能
Restart	Condition 受けた後に処理を継続させる機能

これらは一番プリミティブなレベルでは、クラス定義、`throw`、`catch`、`go` 文などにより実現されていますが、Common Lisp の特徴の一つである Macro を利用して、使い勝手の良い構文が言語仕様として定義されているのです。

2.1 関数仕様の重要性

『関数処理中の通知すべき状態』や『処理継続機能の提供』は関数仕様として定められます。

『引数で渡された値が 0 から 99 の範囲の整数であるかチェックする関数』を考えましょう。ここで関数の仕様としてありそうな考え方をいくつか以下の表にまとめてみました。

引数	考え方 (1)	考え方 (2)	考え方 (3)
0~ 99	TURE	TRUE	TRUE
-1 以下	FALSE	FALSE	FALSE:warning:下限違反
100 以上	FALSE	FALSE	FALSE:warning:上限違反
”abc”(整数以外)	FALSE	error:整数でない	error:整数でない

また、この関数を利用して、単なるチェック機能としてではなく、『必ず 0 から 99 の数字を返す』機能として、引数が 0 から 99 以外の数字の時には再入力させる、もしくはデフォルトの数字に補正して返す、といった応用も考えられます。

このように、チェック関数一つにも様々な考え方や応用があり、関数の機能としてできるかぎりジェネラルな仕様を定めておく必要があります。

つまり、重要なことは、関数実行中に例外的状況が生じたとき、アドホックな処理をするのではなく、例外を `condition` として通知し、それに応じて提供する処理継続の方法 `restart` について定め、関数仕様として明示的に公開しておくことです。

ANSI Common Lisp のドキュメントでも、各関数仕様説明の “Exceptional Situations” の項目には通知する condition が、“Description” の項目には restart に関する仕様が記述されています。

3 Condition システムの使い方

Common Lisp の言語仕様にもとづき、condition システムの使い方について説明します。

3.1 Condition

Condition を通知する側と受け取る側との間では、状態を表現した `condition` クラスのインスタンスがやりとりされます。それらクラスは全て `condition` クラスを継承したクラスとして定義しなければなりません。言語仕様に、いくつかの condition が定義されています。それらの一部を以下にあげます。

クラス	説明
`condition`	condition トップクラス
serious-condition	通知を受け取った側で、何らかの対処を施さなければならない状態
error	エラー
simple-error	`(error "message")` で通知した時に使われるクラス
storage-condition	アプリケーションを続行しがたい、メモリ管理に関する重大な状態
warning	警告
simple-warning	`(warn "message")` で通知した時に使われるクラス
simple-condition	`(signal "message")` で通知した時に使われるクラス

ここで condition のクラス継承というのはとても大事な役割をはたします。3.3.3 章で述べますが、`ignore-errors` マクロを利用すると `error` クラスの condition 通知は無視することができます。「無視されては困る」「無視された場合には続行不能な状態におちいる」例外を通知する際には `error` を継承したクラスを利用すべきではありません。上の表で、`storage-condition` は、例えばメモリが枯渇状態にあり、処理を継続すると更に悪い状況になることが予想されるような場合に通知される condition であり、`ignore-errors` では無視できない condition、即ち、`error` を継承せずに `serious-condition` を継承する condition として定義されています。

3.1.1 Condition の定義と生成 (`define-condition`, `make-condition`)

`condition` クラスは、`define-condition` マクロを利用して定義します。`defclass` マクロと同様のキーワードパラメータ指定の他 `:report` キーワードパラメータの指定が可能で、メッセージの表示方法を指定することができます。また、`condition` クラスのインスタンスの生成には、`make-condition` 関数を利用します。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; condition 発生時刻を記録し、メッセージに付加して表示する condition の定義例
;;;
(define-condition my-condition (simple-condition)
  ((universal-time :initform (get-universal-time)
                   :accessor universal-time))
  (:report (lambda (condition stream)
             (multiple-value-bind (ss mm hh d m y)
                 (decode-universal-time (universal-time condition))
               (format stream "[~D-~2,'0D-~2,'0D ~2,'0D:~2,'0D:~2,'0D] "
                       y m d hh mm ss)
               (call-next-method)))))

CL-USER(180): (error (make-condition 'my-condition
                       :format-control "This condition is my-condition. val=~A."
                       :format-arguments '(10)))

Error: [2006-05-22 19:19:04] This condition is my-condition. val=10.
  [condition type: MY-CONDITION]
```

define-condition はマクロです。上の my-condition の定義は、おおよそ次のようなプログラムにマクロ展開されます。

```
(PROGN (DEFCLASS MY-CONDITION (SIMPLE-CONDITION)
                 ((UNIVERSAL-TIME :INITFORM (GET-UNIVERSAL-TIME) :ACCESSOR
                                  UNIVERSAL-TIME)))
       (DEFMETHOD PRINT-OBJECT ((CONDITION MY-CONDITION) STREAM)
         (IF (OR *PRINT-ESCAPE* *PRINT-READABLY*)
             (CALL-NEXT-METHOD)
           (FUNCALL #'(LAMBDA (CONDITION STREAM)
                        (MULTIPLE-VALUE-BIND (SS MM HH D M Y)
                            (DECODE-UNIVERSAL-TIME
                             (UNIVERSAL-TIME CONDITION))
                          (FORMAT STREAM
                                  "[~D-~2,'0D-~2,'0D ~2,'0D:~2,'0D:~2,'0D] "
                                  Y M D HH MM SS)
                          (CALL-NEXT-METHOD)))
                    CONDITION STREAM)))
       'MY-CONDITION)
```

defclass により my-condition クラスが定義され、:report キーで指定した関数が print-object メソッドとして定義されるマクロであることがわかります。

:report で指定した関数を *condition reporter* と呼び、表示されるメッセージを *condition message* と呼びます。*Condition message* は大文字で開始しピリオドで終わる文章として書くことが望ましいです。

3.2 Signaling

Condition の通知には、error, cerror, signal, warn 関数を利用します。[2]

[2]Unix の世界で signaling と言ったとき 割込み+通知 のことを指すことが多いですが、本来 割込み (interrupt) と 通知 (signal) は直交した概念であり分けて考えるべきです。 Common Lisp における割込みについては 4.3 章でふれます。

3.2.1 error 関数

Condition を通知します。error 関数で通知した場合、その condition が handling されていなければデバッガが起動されます。

2.1 でとりあげた、0 から 99 の範囲の整数であるかチェックする関数を例にあげます。2.1 章の考え方 (2) に相当します。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 引数 val が 整数でない場合、simple-error を通知する関数の例
;;;
(defun check-value-with-error (val)
  (unless (integerp val)
    (error "~A is not a integer number." val))
  (<= 0 val 99))

CL-USER(9): (check-value-with-error 0)     ; 範囲内の整数値を指定する
T
CL-USER(10): (check-value-with-error 3.14) ; 整数以外を指定する
Error: 3.14 is not a integer number.       ; エラーメッセージが表示されデバッガが起動する

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] CL-USER(11): :continue 0               ; 0 番目の restart 処理を選択する
CL-USER(12)
```

起動されたデバッカの中で入力できるコマンドは、Lisp のインプリメンテーションに依存します。いずれのインプリメンテーションにおいても、対話的にコマンドを実行してデバッガから抜けることができます。上の例は Allegro CL (http://franz.com) におけるデバッガの例です。

error 関数の引数には、condition クラスのインスタンス、もしくは文字列で *condition message* を指定することができます。文字列で指定した場合、simple-error クラスのインスタンスが生成されます。

3.2.2 cerror 関数

error 関数と同様 condition を通知し、その condition が handling されていなければデバッガが起動されます。error 関数と違う点としては、restart 名に continue を指定して処理を継続させることが可能です。この restart が選択された場合、cerror 関数は nil を返して、condition 通知のあった以降の処理が継続されます。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 引数 val が 0 から 99 の範囲の数字になるまで、再入力させる関数の例
;;;     値が範囲内になったらその値を返す
;;;
(defun check-value-with-cerror (val)
  (loop                   ; 範囲内の値になるまで loop を続ける
      if (integerp val)    ; val の値を検査する if 文 block
        if (< val 0) do
          (cerror "Enter a valid integer number." "~D is too small." val)
          (format t "~&Input a bigger number: ")
        else if (> val 99) do
          (cerror "Enter a valid integer number." "~D is too big." val)
          (format t "~&Input a smaller number: ")
        else do
             (return val)   ; 範囲内ならその値を返す
        end
      else do
        (cerror "Enter a integer number." "~A is not a integer number." val)
        (format t "~&Input a integer number: ")
      do                   ; val が範囲外だった場合の、値の再入力機能
        (setq val (read))
        (fresh-line)))

;;; デバッガ内で対話的に処理を継続させた例

CL-USER(25): (check-value-with-cerror -1)
Error: -1 is too small.

Restart actions (select using :continue):
 0: Enter a valid integer number.            ; cerror で指定した選択肢も表示される
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1c] CL-USER(26): :continue 0                ; cerror の選択肢 0 を選択する
Input a bigger number: 100                   ; 処理が継続し、値の入力が要求される
Error: 100 is too big.                       ; 再チェック後またエラーが発生した

Restart actions (select using :continue):
 0: Enter a valid integer number.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1c] CL-USER(27): :continue 0
Input a smaller number: 33                   ; 範囲内の値を設定するとその値が返る
33
CL-USER(26):

;;; continue を使用して restart させた例

CL-USER(28): (handler-bind ((error
                             #'(lambda (cond)
                                 (declare (ignore cond))
                                 (continue)))) ; cerror 関数を終了し処理を継続させる
               (check-value-with-cerror 3.14))
Input a integer number: -1                     ; デバッガは起動されず処理が継続する
Input a bigger number: 100
Input a smaller number: 33
33
CL-USER(29):
```

3.2.3 signal 関数

Condition を通知します。error, cerror 関数とは異なり、signal 関数ではデバッガは起動されず、設定されている全てのハンドラーを実行した後 nil を返して処理が継続されます。

無視されても構わない状態の通知に使います。用途としては様々ありますが、ここでは関数が呼ばれた時のタイムスタンプとして利用する例をあげます。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 先の error 関数の例を改造し、my-condition を利用してタイムスタンプを打つ
;;;
(defun check-value-with-error-2 (val)
  (signal (make-instance 'my-condition
                         :format-control "Called check-value-with-error."))
  (unless (integerp val)
    (error "~A is not a integer number." val))
  (<= 0 val 99))

;;; Handling している例

CL-USER(30): (handler-bind ((my-condition   ; condition をハンドリングしてプリントする
                              #'(lambda (cond)
                                  (format t "~A~%" cond))))
               (check-value-with-error-2 50)
               (sleep 5)
               (check-value-with-error-2 50))
[2006-07-25 19:15:30] Called check-value-with-error.
[2006-07-25 19:15:35] Called check-value-with-error.
T
CL-USER(31):

;;; Handling していない例

CL-USER(16): (check-value-with-error-2 50)  ; signal の通知では何も起きない
T
CL-USER(17):
```

signal 関数の引数には、通知したい condition クラスのインスタンスあるいはクラス名を、もしくは文字列で *condition message* を指定することができます。文字列で指定した場合、simple-condition クラスのインスタンスが生成されます。

3.2.4 warn 関数

Warning を通知します。引数には warning クラスのインスタンスあるいはクラス名を、もしくは文字列で *condition message* を指定することができ、文字列を指定した場合はsimple-warning クラスのインスタンスが生成されます。warning クラスでないインスタンスを指定した場合は type-error のエラーが発生します。

warn 関数ではデバッガは起動されませんが、設定されている全てのハンドラーを実行した後 *error-output* stream に *condition message* を出力します。その後、nil を返して処理が継続されます。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 引数 val が 0 から 99 の範囲の整数でない場合、simple-warning を通知する関数の例
;;;
(defun check-value-with-warn (val)
  (if (integerp val)
      (cond ((< val 0)
             (warn "~D is too small." val)
             (format t "Here is in check-value.~%")
             nil)
            ((> val 99)
             (warn "~D is too big." val))
            (t
             t))
    (error "~A is not a integer number." val)))

;;; Handling していない例

CL-USER(242): (check-value-with-warn -10)
Warning: -10 is too small.     ; Warning メッセージが表示され、
Here is in check-value.        ; warn 関数は終了し、処理が継続される
NIL

;;; Handling している例

CL-USER(231): (handler-bind ((simple-warning
                             #'(lambda (cond)
                                 (declare (ignore cond))
                                 (format t "Here is in handler.~%"))))
               (check-value-with-warn -10))
Here is in handler.          ; ハンドラーが実行され、
Warning: -10 is too small.   ; ハンドラー実行後に warning メッセージが表示される
Here is in check-value.      ; 表示後 warn 関数は終了し、処理が継続される
NIL

;;; ハンドラーの中で、以降のハンドラーを実行させない例
;;;     muffle-warning により、以降のハンドラーを実行せずに warn 関数を終了させて処理を
;;;     継続させる。当然この場面においても unwind-protect の後処理は必ず実行される。

CL-USER(232): (handler-bind ((simple-warning
                             #'(lambda (cond)
                                 (declare (ignore cond))
                                 (unwind-protect
                                     (progn
                                       (format t "Here is in handler.~%")
                                       (muffle-warning))
                                   (format t "Handler finished.~%")))))
                (check-value-with-warn -10))
Here is in handler.          ; ハンドラーが実行され、
Handler finished.            ; (unwind-protect の後処理が実行されている)
Here is in check-value.      ; muffle-warning により以降のハンドラーは実行されないため、
NIL                          ; warning メッセージは表示されない
```

3.2.5 デバッガの強制起動

signal 関数と warn 関数による通知ではデバッガは起動されませんが、スペシャル変数*break-on-signals* を利用して強制的に起動することができます。発生した *condition* に対して (typep *condition* *break-on-signals*) が true の場合、つまり、*break-on-signals* に指定したクラスもしくはそのサブクラスが signaling された時、デバッガが起動されます。

デバッガは signaling の直後、ハンドラーの実行直前に起動されます[3]。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; warn 関数による通知で強制的にデバッガを起動する例
;;;
CL-USER(36): (let ((*break-on-signals* 'warning))
                   (check-value-with-warn -10))
Break: -10 is too small.                    ; warn による通知だがデバッガが起動される
break entered because of *break-on-signals*.

Restart actions (select using :continue):
 0: return from break.
 1: skip warning.
 2: continue processing.
 3: Return to Top Level (an "abort" restart).
 4: Abort entirely from this (lisp) process.
[1c] CL-USER(37): :cont 1                   ; warn 関数を終了し、処理を継続させる
Here is in check-value.
NIL
```

[3] Allegro CL では、この時のデバッガ起動に break 関数を利用して実現しています。break 関数についてはこの例の後にあげた例を参照してください。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 全てのクラスの親クラスである t を指定して実行した例
;;;
CL-USER(3): (let ((*break-on-signals* t))
              (handler-bind ((my-condition
                              #'(lambda (cond)
                                  (format t "My-condition handler with ~S.~%" cond)))
                             (error
                              #'(lambda (cond)
                                  (format t "Error handler with ~S.~%" cond))))
                (signal (make-condition 'my-condition
                          :format-control "This is my-condition."))
                (error "This is simple-error.")))
Break: [2006-08-02 12:30:32] This is my-condition.      ; signal 通知でデバッガが起動
break entered because of *break-on-signals*.

Restart actions (select using :continue):
 0: return from break.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1c] CL-USER(4): :cont 0                                ; 処理の継続を選択
My-condition handler with #<MY-CONDITION @ #x40c61992>. ; ハンドラーが実行される
Break: This is simple-error.                            ; error 通知でデバッガが起動
break entered because of *break-on-signals*.

Restart actions (select using :continue):
 0: return from break.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1c] CL-USER(5): :cont 0                               ; 処理の継続を選択
Error handler with #<SIMPLE-ERROR @ #x40c6641a>.       ; ハンドラーが実行され、
Error: This is simple-error.                           ; error 関数による通知のため、
                                                       ; 最後にデバッガが起動する
Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] CL-USER(6): :cont 0
CL-USER(7):
```

また、関数のある場所で強制的にデバッガを起動したい場合 (break point の設定) には、break 関数を利用して signaling します[4]。Restart 名 continue を指定して処理を継続することができます。この時 break 関数は nil を返します。

[4] Allegro CL では、simple-condition を親クラスとした excl:simple-break クラスのインスタンスを生成して signaling しています。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; break 関数による通知でデバッガを起動する例（break point の設定）
;;;
(defun foo-with-break ()
  (format t "Start foo-with-break~%")
  (break)
  (format t "End foo-with-break~%"))
CL-USER(40): (foo-with-break)
Start foo-with-break
Break: call to the ‘break’ function.    ; break 関数の呼出しによりデバッガが起動

Restart actions (select using :continue):
 0: return from break.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1c] CL-USER(41): :cont 0               ; break を終了させ、処理を継続させる
End foo-with-break
nil
CL-USER(42):
```

また、condition を引数とする invoke-debugger 関数を利用してデバッガを起動することもできます。break 関数は singlaing を行なうのに対し、invoke-debugger は signaling をせずにデバッガを起動するという違いがあります。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; warning のハンドラーでデバッガを起動する例
;;;
CL-USER(117): (handler-bind ((simple-warning #'invoke-debugger))
                (check-value-with-warn -10))
Debug: -10 is too small.
  [condition type: SIMPLE-WARNING]

Restart actions (select using :continue):
 0: skip warning.
 1: continue processing.
 2: Return to Top Level (an "abort" restart).
 3: Abort entirely from this (lisp) process.
[1] CL-USER(118):
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; break 関数と invoke-debugger 関数の違い
;;;
;;; break は signaling するため、handling できる
CL-USER(242): (handler-case
                  (break)
                (simple-break (cond)
                  (format t "~&~S~%~A" cond cond)))
#<SIMPLE-BREAK @ #x75abe772>
call to the ‘break’ function.
NIL
CL-USER(243):

;;; invoke-debugger はデバッガを起動するだけであり、handling とは無関係

CL-USER(243): (handler-case
                 (invoke-debugger 'simple-break)
               (simple-break (cond)
                 (format t "~&~S~%~A" cond cond)))
Debug: #<SIMPLE-CONDITION @ #x75abf81a>
  [condition type: SIMPLE-CONDITION]

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] CL-USER(244): :cont 0
CL-USER(245):
```

デバッガが起動される直前に実行する関数を、スペシャル変数*debugger-hook* で指定することもできます。次の例は ANSI Common Lisp のドキュメント *debugger-hook* に載っている例です。この中では 3.5.2 章で説明する機能 invoke-restart-interactively を利用しています。こちらの章も参照してください。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 独自のデバッガを作成する例（*debugger-hook* のドキュメントより引用）
;;;
(defun one-of (choices &optional (prompt "Choice"))
  (let ((choice-no 0))
    ;; restart 選択肢を標準出力に表示する
    (loop for choice in choices
        as no from 1
        do (format t "~&[~D] ~A: " no choice))
    ;; ユーザに restart 番号を要求する
    (loop until (typep choice-no `(integer 1 ,(length choices)))
        do (format t "~&~A: " prompt)
           (setq choice-no (read))
           (fresh-line))
    ;; 選択された restart を返す
    (nth (- choice-no 1) choices)))

(defun my-debugger (condition me-or-my-encapsulation)
  (format t "~&Fooey: ~A" condition)            ; エラーメッセージを表示する
  (let ((restart (one-of (compute-restarts))))   ; restart を表示し選択させる
    (when (not restart)
      (error "My debugger got an error."))
    (let ((*debugger-hook* me-or-my-encapsulation))
      (invoke-restart-interactively restart))))  ; 選択された restart を実行する

CL-USER(8): (let ((*debugger-hook* #'my-debugger))
              (check-value-with-cerror -1))
Fooey: -1 is too small.
[1] Enter a valid integer number.:
[2] Return to Top Level (an "abort" restart).:
[3] Abort entirely from this (lisp) process.:
Choice: 4                            ; 表示されていない選択肢を指定する
Choice: 1                            ; 存在しない選択肢が入力されたため、再要求された
Input a bigger number: 10            ; 値を再設定する restart を選択し、実行された
10
CL-USER(9):
```

3.3 Handling

Signaling を監視し、condition を受け取った時に実行する処理 (『ハンドラー』と呼びます) を設定します。

ハンドラーは階層的に設定され、ハンドラーの中で明示的に restart を実行したり (3.5 章参照)、`throw`, `catch`, `go` 文などを使って処理がジャンプされていない限り、signaling された condition は toplevel まで通知されます。そして toplevel では、通知 condition に対応した処理 (デバッガの起動や warning メッセージの表示) が行なわれます。

ハンドラーの設定には、次の 3 つのマクロを利用できます。

handler-bind	condition を受け取った時の処理を指定する。 通知された condition は更に上位 (次) のハンドラーにも通知され、そのハンドラーも実行される。
handler-case	condition を受け取った時の処理を指定する。 通知された condition は上位 (次) のハンドラーには通知されず、`handler-case` を終了して処理が継続される。
ignore-errors	error クラスかそれを継承したクラスの condition を受け取った時、その condition を handling して nil を返す。 通知された condition は上位のハンドラーには通知されず、`ignore-errors` を終了して処理が継続される。

後の説明におけるハンドラーの実行順序については、図 1, 図 2 も合わせて参照してください。

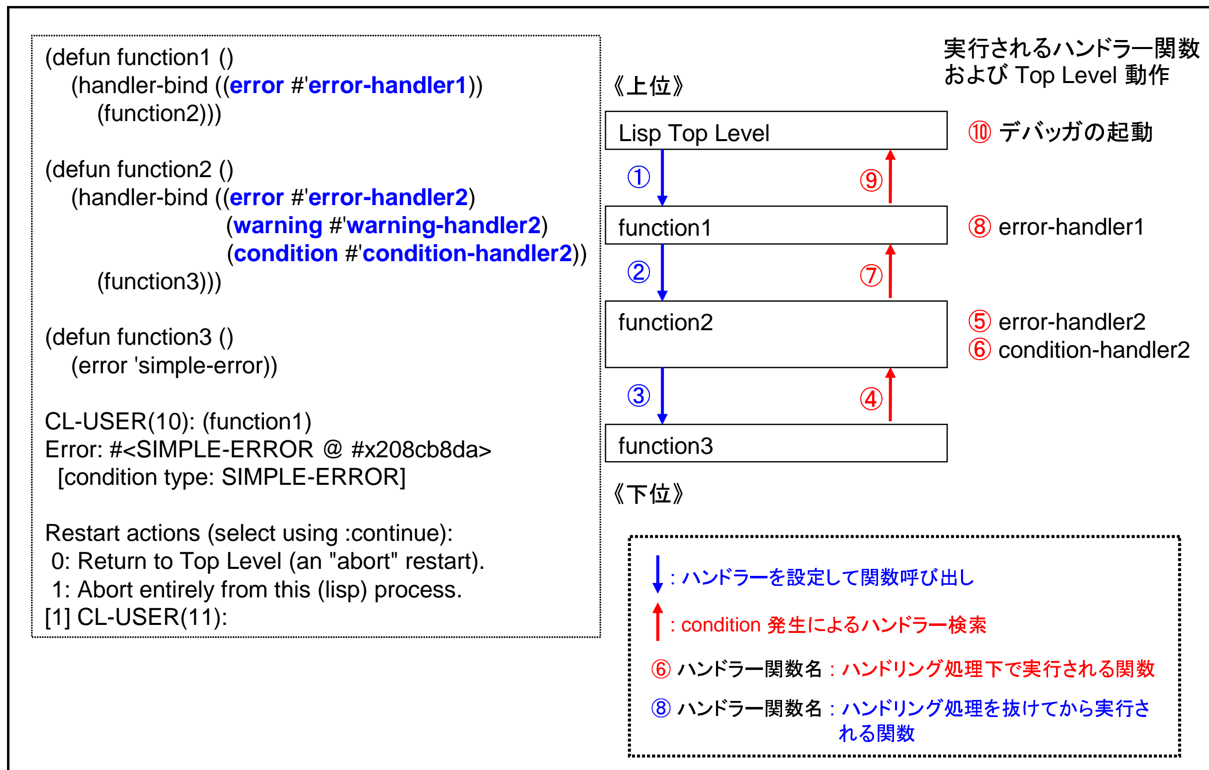

図 1: ハンドラーの中で処理がジャンプしていない場合のハンドラーの実行順序

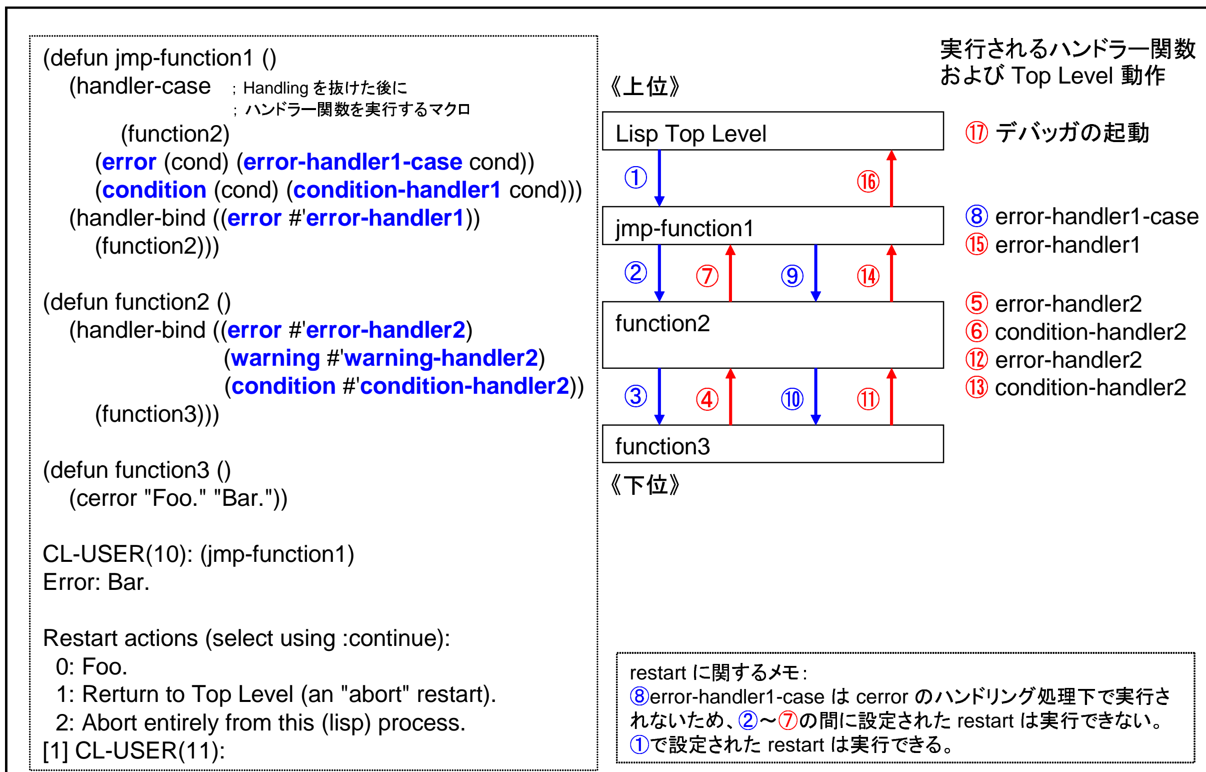

図 2: handler-case **により処理がジャンプしている場合のハンドラーの実行順序**

3.3.1 handler-bind **マクロ**

記述例[5]:

```
(handler-bind ((<condition クラス-A> <ハンドラー-A>)
               (<condition クラス-B> <ハンドラー-B>)
                ...
              )
  <form>)
```

<form> **実行中に指定した** <condition **クラス**> **かそれを継承したクラスの** signaling **があった時、対応する**<**ハンドラー**> **を実行します。**

同じ handler-bind **の中では先に記述したハンドラー** (**リストの左側**) **が優先で、先に実行されます。その後、更に上位で設定されているハンドラーが実行されます。次の例は** simple-error **クラスを受け取った時の例で、**error **クラス**, simple-error **クラス**, condition **クラスのハンドラーが順番に実行されることがわかります。**

[5]**機能の概要説明のため、"記述例" としてフォーマットをあげました。正確な** syntax **は** ANSI Common Lisp **のドキュメントを参照してください。**

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 同じ handler-bind の中に、該当するハンドラーがいくつも設定されている例
;;;
CL-USER(9): (handler-bind ((error
                            #'(lambda (cond)
                                (declare (ignore cond))
                                (format t "Here is in error handling.~%")))
                           (simple-error
                            #'(lambda (cond)
                                (declare (ignore cond))
                                (format t "Here is in simple-error handling.~%")))
                           (warning
                            #'(lambda (cond)
                                (declare (ignore cond))
                                (format t "Here is in warning handling.~%")))
                           (condition
                            #'(lambda (cond)
                                (declare (ignore cond))
                                (format t "Here is in condition handling.~%"))))
              (error "Foo."))  ; simple-error クラスを通知する
Here is in error handling.
Here is in simple-error handling.
Here is in condition handling.
Error: Foo.          ; 優先順に該当するハンドラーが実行され、最後にデバッガが起動する

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] CL-USER(10):
```

3.3.2 handler-case マクロ

記述例:

```
(handler-case
    <form>
  (<condition クラス-A> (<cond>)
    <ハンドラー-A>)
  (<condition クラス-B> (<cond>)
    <ハンドラー-B>)
  ...)
```

<form> 実行中に指定した <condtion クラス> かそれを継承したクラスの signaling があった時、対応する<ハンドラー> を実行します。受け取った condition は <cond> にバインドされ、<ハンドラー> の中で利用できます。

同じ handler-case の中では先に記述したハンドラー (リストの左側) が優先します。

handler-bind と違う点として、対応する <ハンドラー> を実行した後 handler-case は終了します。優先度の低いハンドラーや上位で設定されているハンドラーは実行されません。ハンドラーが実行された時の handler-case の戻り値は <ハンドラー> の値になります。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 同じ handler-case の中に、該当するハンドラーがいくつも設定されている例
;;;
CL-USER(12): (handler-case
                 (progn
                   (format t "Signaling error.~%")
                   (error "Foo."))  ; simple-error クラスを通知する
               (error (cond)
                 (declare (ignore cond))
                 (format t "Here is in error handling.~%")
                 t)
               (simple-error (cond) ; error クラスのハンドラーが実行されるため
                                    ; このハンドラーが実行されることはない
                 (declare (ignore cond))
                 (format t "Here is in simple-error handling.~%")
                 nil))
Signaling error.
Here is in error handling. ; 上位のハンドラーは実行されないため、デバッガも起動されない
T                          ; ハンドラーの値が handler-case の戻り値になる
CL-USER(13):
```

handler-case は handler-bind を利用して定義されたマクロです。上の例はおおよそ次のようなプログラムにマクロ展開されます。

```
(CATCH '#:|Tag17|
  (LET ((#:G1000 NIL))
    (DECLARE (IGNORABLE #:G1000))
    (TAGBODY
        (HANDLER-BIND ((ERROR
                        #'(LAMBDA (EXCL::TEMP)
                             (DECLARE (IGNORABLE EXCL::TEMP))
                             (SETQ #:G1000 EXCL::TEMP)
                             (GO #:G1001)))
                       (SIMPLE-ERROR
                        #'(LAMBDA (EXCL::TEMP)
                             (DECLARE (IGNORABLE EXCL::TEMP))
                             (SETQ #:G1000 EXCL::TEMP)
                             (GO #:G1002))))
          (THROW '#:|Tag17|
            (PROGN (FORMAT T "Signaling error.~%") (ERROR "Foo."))))
      #:G1001
        (THROW '#:|Tag17|
          (LET ((COND #:G1000)) (FORMAT T "Here is in error handling.~%") T))
      #:G1002
        (THROW '#:|Tag17|
          (LET ((COND #:G1000))
            (FORMAT T "Here is in simple-error handling.~%")
            NIL)))))
```

これを見てわかるように、handler-case のハンドラーは handler-bind のハンドラーから go 文で抜け出した後に実行されます。このため、handler-case のハンドラーの中では <form> 以下で設定されている restart を実行することはできません。上位で設定された restart であればその restart の設定下にあるため、実行することができます。先にあげた図 2 も参照してください。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; handler-case の ハンドラーで、下位で設定された restart を実行してエラーになる例
;;;
CL-USER(206): (handler-case
                  (check-value-with-warn -1)
                (warning (cond)
                  (declare (ignore cond))
                  (format t "Here is in handler-case~%")
                  (muffle-warning)))     ; warn 関数を終了させ処理を継続させる
Here is in handler-case
Error: a restart named muffle-warning is not active.
  [condition type: CONTROL-ERROR]        ; Restart が設定されていない旨のエラーが上がる

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] CL-USER(207):

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; handler-case の ハンドラーで、上位で設定された restart を実行する例
;;;
CL-USER(215): (defun foo (val)
                (handler-case
                    (check-value-with-warn val)
                  (warning (cond)
                    (declare (ignore cond))
                    (format t "Here is in handler-case~%")
                    (invoke-restart 'my-restart 7))))  ; restart を実行させる
FOO
CL-USER(216): (restart-case
                  (foo -1)
                (my-restart (&optional v) (+ v 2)))
Here is in handler-case
9                             ; (+ 7 2)
CL-USER(217):
```

3.3.3 ignore-errors マクロ

記述例:

```
(ignore-errors <form>)
```

<form> 実行中に error クラスかそれを継承したクラスの signaling があった時、nil を返します。通知された error クラスのインスタンスは、2 値目の戻り値として返します[6]。

エラーが通知された場合には nil として扱えば十分な場合に利用します。

[6] Common Lisp は、関数の戻り値として複数の値 (多値) を返すことができます。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; val が範囲外の場合は default 値を返す関数の例
;;;
(defun foo (val)
  ;; 範囲外の場合は default 値 100 を返す関数
  (if (ignore-errors (check-value-with-error val))
      (* 10 val)
    100)) ; default 値
CL-USER(46): (foo 3.14) ; 発生した error は ignore-errors でハンドリングされ nil になる
100                     ; default 値 100 が返る
CL-USER(47):
```

ingore-errors は次の記述と同値です。

```
(handler-case
  (progn
    <form>)
  (error (cond)
    (values nil cond)))
```

ignore-errors は error クラスかそれを継承したクラスの通知をハンドリングして無視するものであり、通知が error 関数によるものか signal 関数によるものかとは関係がありません。混同しないようにしましょう。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; ignore-errors は error クラスに対するハンドリングであり、signaling の関数とは
;;; 無関係であることを示す例
;;;

;;; error による通知
;;; error object を catch して、その object が 2 値目で返る。デバッガは起動されない

CL-USER(40): (ignore-errors (error 'simple-error))
NIL
#<SIMPLE-ERROR @ #x410efc22>

;;; signal による通知でも、error object を catch して、その object が 2 値目で返る

CL-USER(41): (ignore-errors (signal 'simple-error))
NIL
#<SIMPLE-ERROR @ #x410f02ea>

;;; storage-condition は error クラスの子クラスではない。
;;; このため、ignore-errors ではハンドリングされず、単に signal 関数の戻り値 nil が返る

CL-USER(42): (ignore-errors (signal 'storage-condition))
NIL

;;; error 関数による通知のため、デバッガが起動される

CL-USER(43): (ignore-errors (error 'storage-condition))
Error: #<STORAGE-CONDITION @ #x410f105a>
  [condition type: STORAGE-CONDITION]

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] CL-USER(44):
```

3.3.4 default handler を使う

ある signaling に対する対処を一旦上位のハンドラーにゆだね、上位で restart や go 文による処理のジャンプが行なわれなかった時、自分で設定した処理を実行するようコーディングする方法があります。これを default handler と呼びます。次のように使います。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; default handler を利用した例
;;;     check-value-with-error で error が発生した時、一旦上位に対処をゆだねる。
;;;     signal 関数により上位に通知しているため、上位で restart が実行されなかった場合
;;;     は処理が戻ってくる。その時は abort させる。
;;;
(defun check-value-with-default-handler (val)
  (handler-bind ((error #'(lambda (cond)
                            (signal cond)   ; signal 関数で一旦上位に通知する
                            (format t "Here is in default handler.~%")
                            (abort))))      ; 処理を止める
    (check-value-with-error val)))

;;; default hadler が実行される場合の例

CL-USER(153): (check-value-with-default-handler 3.14)
Here is in default handler.
CL-USER(154):

;;; 上位で handler-case でハンドリングしている例
;;; handler-case により処理がジャンプするため、default handler は実行されない。

CL-USER(155): (handler-case
                  (check-value-with-default-handler 3.14)
                (error (cond)
                  (declare (ignore cond))
                  (format t "I get an error. Return a default value now.~%")
                  100))
I get an error. Return a default value now.
100
CL-USER(156):
```

3.4 Restart の設定

ある処理から restart を設定した処理へと、ダイナミックに処理を移動させる機能を提供します。これを利用して、Signaling の原因となった問題を解決させ、処理を継続させる機能を提供することができます。

3.2.2 章の cerror の例でも値を設定しなおして処理を継続させる例は示しましたが、restart の選択肢は 1 つ増やせただけですし、また、cerror 関数を終了させて処理を継続させることしかできないため、signaling 箇所と継続開始箇所は同じでした。Restart 機能を利用すると複数の選択肢を提供でき、また、signaling 箇所と継続開始箇所を別にすることも可能になり、更に柔軟な機能を提供することができます。

ダイナミックに処理を移動してくることができる機能restart の設定には、restart-bind, restart-case, with-simple-restart マクロを利用します。

restart-bind は primitive なものという位置付けで主に他の restart マクロから使われ、直接的にはあまり使われません。with-simple-restart は restart-case を使ったマクロです。

この章では、プログラマが直接利用する機会が多い restart-case について説明し、restart-case に比べてより柔軟な restart-bind については例だけをあげます。また、説明はしませんが、signaling と restart 設定を関連付けるための with-condition-restarts マクロがあります。Restart 設定に関連するマクロであることは記憶しておきましょう。

3.4.1 restart-case マクロ

記述例:

```
(restart-case
    <restartable-form>
  (<case-name-A> <parameter-list>    ; restart 名 および restart 処理のパラメタ名
    :interactive <interactive-expression-A>  ; 対話機能 (省略可)
    :report <report-expression-A>    ; Restart 表示メッセージ (省略可)
    :test <test-expression-A>        ; 有効な restart かどうかの判定関数 (省略可)
    <restart-form-A>)                ; restart 処理
  (<case-name-B>
    ...)
  ...)
```

<restartable-form> 実行中、restart 名<case-name> が指定されて invoke-restart 関数 (3.5.1 章参照) が呼ばれた時、呼ばれた箇所から<restart-form>にダイナミックに処理を移動するよう設定します。<restartable-form> 内での signaling を受けた上位ハンドラーからも、同様に移動してくることができます。

同じ restart-case の中では先に記述された restart (リストの左側) が優先し、入れ子の場合は後で記述された restart (内側) が優先します。

<parameter-list> には、対話的に restart が実行された場合は :interactive に指定した関数の戻り値が、invoke-restart により非対話的に restart が実行された場合は invoke-restart 関数で指定した <arguments> が適用されます。

:test に指定した関数の戻り値が t の場合に、この restart 設定は有効になります (省略時は t です)。

以下はデバッガによる対話的な restart の例です。invoke-restart による非対話的な restart の例は 3.5.1 章を参照してください。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; val が 0 から 99 の範囲の数字でない場合、値もしくは範囲を再設定する restart の例
;;;
(defun check-value-with-restart (val)
  (loop with low = 0   ; 下限初期値
      with high = 99   ; 上限初期値
      if (and (integerp val)
              (<= low val high)) return
        (values val low high)            ; 範囲内の場合、結果として多値を返す
      else do          ; 範囲外の場合の処理
           (restart-case
               (error "Illegal value: ~D." val)
             ;; チェック値を変更する restart の設定
             (store-value (new-val)
                 :report "Set new value."
                 :interactive (lambda ()
                                (format t "Enter a new value: ")
                                (list (read)))
               (setq val new-val))
             ;; 範囲を変更する restart の設定
             (set-new-limit (new-low new-high)
                 :report "Set new lower value."
                 :interactive (lambda ()
                                (let (new-low new-high)
                                  (format t "Enter a new lower: ")
                                  (setq new-low (read))
                                  (format t "Enter a new higher: ")
                                  (setq new-high (read))
                                  (list new-low new-high)))
               (setq low new-low)
               (setq high new-high)))))


;;; 値を再設定する restart の実行例

CL-USER(42): (check-value-with-restart -1)
Error: Illegal value: -1.
  [condition type: SIMPLE-ERROR]

Restart actions (select using :continue):
 0: Set new value.
 1: Set new lower value.
 2: Return to Top Level (an "abort" restart).
 3: Abort entirely from this (lisp) process.
[1] CL-USER(43): :cont 0       ; 値の再設定を選択する
Enter a new value: 10          ; 値を入力する
10      ; 変更したチェック値
0       ; 下限
99      ; 上限
CL-USER(44):
```

```
;;; 範囲を再設定する restart の実行例

CL-USER(44): (check-value-with-restart -1)
Error: Illegal value: -1.
  [condition type: SIMPLE-ERROR]

Restart actions (select using :continue):
 0: Set new value.
 1: Set new lower value.
 2: Return to Top Level (an "abort" restart).
 3: Abort entirely from this (lisp) process.
[1] CL-USER(45): :cont 1      ; 範囲の再設定を選択
Enter a new lower: -5         ; 下限を入力する
Enter a new higher: 50        ; 上限を入力する
-1     ; チェック値
-5     ; 変更した下限
50     ; 変更した上限
CL-USER(46):
```

この例の中で、:interactive で new-low などの別の変数に再入力された値を一旦設定し、その後 restart 処理の中で setq していることは、一見まどろこしいコードに見えます。:interactive の中で setq するだけで済むように思えますが、そのようにコーディングすると、:interactive は対話的な restart の場合しか呼ばれないため、非対話的な restart では setq が実行されないことになります。よって、restart 処理の中での setq は外すことはできないのです。

エンドユーザに対する interface としてデバッガを利用し、restart 機能を利用して無限に処理を続ける対話システムを提供することができるようになります。既存の対話機能 (デバッガ) を利用することで、対話処理の開発コストを下げることができます。ただし、デバッガの中ではプログラムを変更するなど、リスナーからできることは何でもできるので、注意が必要です[7]。

[7]セキュリティ面をしっかりさせるのなら先の *debugger-hook* の例のように、独自のデバッガを提供した方が良いでしょう。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; デバッガを入力 interface として、無限に平方根を求め続ける例
;;;
(defun my-sqrt-loop ()
  (loop with val = 0
      do (restart-case
             (let ((result (sqrt val)))
               (format t "sqrt ~4D = ~,6F~%" val result)
               (break "What do you want to do?"))
           ;; 値を再入力させる restart の設定
           (store-value (new-value)
               :report "Set new value."
               :interactive (lambda ()
                              (format t "Enter a new number: ")
                              (list (read)))
             (setq val new-value))
           ;; 1 足した値を利用する restart の設定
           (add-1 ()
               :report "Increment value."
             (incf val))
           ;; 1 引いた値を利用する restart の設定
           (delete-1 ()
               :report "Decrement value."
             (decf val)))))
```

```
CL-USER(73): (my-sqrt-loop)
sqrt    0 = 0.000000
Break: What do you want to do?

Restart actions (select using :continue):
 0: return from break.
 1: Set new value.
 2: Increment value.
 3: Decrement value.
 4: Return to Top Level (an "abort" restart).
 5: Abort entirely from this (lisp) process.
[1c] CL-USER(74): :cont 1              ; 値を指定して sqrt 計算することを選択する
Enter a new number: 10
sqrt   10 = 3.162278
Break: What do you want to do?

Restart actions (select using :continue):
 0: return from break.
 1: Set new value.
 2: Increment value.
 3: Decrement value.
 4: Return to Top Level (an "abort" restart).
 5: Abort entirely from this (lisp) process.
[1c] CL-USER(75): :cont 2              ; 前の値に 1 足した値で計算することを選択する
sqrt   11 = 3.316625
Break: What do you want to do?

Restart actions (select using :continue):
 0: return from break.
 1: Set new value.
 2: Increment value.
 3: Decrement value.
 4: Return to Top Level (an "abort" restart).
 5: Abort entirely from this (lisp) process.
[1c] CL-USER(76): :cont 3              ; 前の値から 1 引いた値で計算することを選択する
sqrt   10 = 3.162278
Break: What do you want to do?

Restart actions (select using :continue):
 0: return from break.
 1: Set new value.
 2: Increment value.
 3: Decrement value.
 4: Return to Top Level (an "abort" restart).
 5: Abort entirely from this (lisp) process.
[1c] CL-USER(77): :cont 4              ; 終了
CL-USER(78):
```

3.4.2 restart-bind マクロ

3.3.2 章で述べた handler-case のハンドラーがハンドリング処理を抜けてから実行されるのと同様、restart-case は restart 設定下を抜けてから restart 処理が実行されます。つまり、同じ restart-case 内の restart を呼ぶことはできません[8]。

[8] 3.5.3 章の compute-restarts を restart 処理の中で表示させ、確認してみましょう。

これに対し、restart-bind は restart 設定下で実行される restart 処理を記述することができます。これを利用した例をあげます。サンプル関数内では 3.5.2 章で説明する invoke-restart-interactively を、実行例では 3.5.1 章で説明する invoke-restart を利用しています。これらの章を読んでから、この例を理解してください。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 関数の不動点探索の例
;;;     不動点: x が f(x) = x を見たす時、x を関数 f の不動点 (fixed point) と呼ぶ。
;;;             ある関数 f について、最初の予測値から始め、f を値があまり変わらなくな
;;;             るまで繰り返し作用させせることで、不動点を見付ける。
;;;     参考文献: 『計算機プログラムの構造と解釈』
;;; 関数 function の不動点を予測値 init-guess から計算を始めて不動点を求める。
;;; 変化が tolerance より小さくなった値を不動点とする。
;;; 計算回数は最大 limit 回とする。
;;; 提供する restart: change-function => 関数と予測値を設定しなおす
;;;                   change-guess    => 予測値を設定しなおす
;;;                   figure-more     => 計算回数を増やす
(defun find-fixed-point (func init-guess tolerance limit)
  (loop as guess = init-guess then next
      as count from 0
      as next = (funcall func guess)    ; f(x) を求める
      as sub = (abs (- guess next))     ; 変化を求める
      when (or (< sub tolerance)        ; 変化が許容範囲以下もしくは最大回数の時
               (>= count limit)) do
        (format t "~&The fixed point of ~A is ~,6F.~%" func next)
        (format t "  [guess=~,2F, count/limit=~D/~D, sub/tolerance=~,6F/~,6F]~%"
                init-guess count limit sub tolerance)
        (block my-tag
          (restart-bind ((change-function #'(lambda (new-func)
                                              (setq func new-func)
                                              (invoke-restart-interactively 'change-guess))
                            :report-function
                              #'(lambda (stream)
                                  (format stream "Eval another function." func))
                            :interactive-function
                              #'(lambda ()
                                  (format t "Enter a new func: ")
                                  (list (read))))
                        (change-guess #'(lambda (new-guess)
                                          (setq init-guess new-guess)
                                          (setq next new-guess)
                                          (setq count 0)
                                          (return-from my-tag t))
                            :report-function
                              #'(lambda (stream)
                                  (format stream "Try to figure another guess. [guess=~,2F] "
                                          init-guess))
                            :interactive-function
                              #'(lambda ()
                                  (format t "Enter a new guess: ")
                                  (list (read))))
                        (figure-more #'(lambda (more)
                                         (setq limit (+ count more))
                                         (return-from my-tag t))
                            :report-function
                              #'(lambda (stream)
                                  (format stream "Try to figure more. [count=~D] " count))
                            :interactive-function
                              #'(lambda ()
                                  (format t "How many time?: ")
                                  (list (read)))))
            (break "What do you want to do?")))))
```

```
;;; 余弦関数（cos）の不動点を求める

CL-USER(197):  (find-fixed-point 'cos 1.0 0.00001 20)
The fixed point of COS is 0.739018.
  [guess=1.00, count/limit=20/20, sub/tolerance=0.000166/0.000010]
Break: What do you want to do?        ; sub によると、まだ収束していない

Restart actions (select using :continue):
 0: return from break.
 1: Eval another function.
 2: Try to figure another guess. [guess=1.00]
 3: Try to figure more. [count=20]
 4: Return to Top Level (an "abort" restart).
 5: Abort entirely from this (lisp) process.
[1c] CL-USER(198): :cont 3            ; あと 15 回計算させる
How many time?: 15
The fixed point of COS is 0.739082.
  [guess=1.00, count/limit=28/35, sub/tolerance=0.00007/0.000010]
Break: What do you want to do?        ; 28 回の計算で収束した

Restart actions (select using :continue):
 0: return from break.
 1: Eval another function.
 2: Try to figure another guess. [guess=1.00]
 3: Try to figure more. [count=28]
 4: Return to Top Level (an "abort" restart).
 5: Abort entirely from this (lisp) process.
[1c] CL-USER(199): :cont 2            ; 予測値を変えてみる
Enter a new guess: 1.5
The fixed point of COS is 0.739082.
  [guess=1.50, count/limit=31/35, sub/tolerance=0.000007/0.000010]
Break: What do you want to do?        ; 31 回の計算で収束した

Restart actions (select using :continue):
 0: return from break.
 1: Eval another function.
 2: Try to figure another guess. [guess=1.50]
 3: Try to figure more. [count=31]
 4: Return to Top Level (an "abort" restart).
 5: Abort entirely from this (lisp) process.
[1c] CL-USER(200): :cont 1            ; 関数を変えてみる。（sqrt 5）を求める式である
Enter a new func: (lambda (y) (/ (+ y (/ 5 y)) 2))
Enter a new guess: 1.0
The fixed point of (LAMBDA (Y) (/ (+ Y (/ 5 Y)) 2)) is 2.236068.
  [guess=1.00, count/limit=5/35, sub/tolerance=0.000001/0.000010]
Break: What do you want to do?

Restart actions (select using :continue):
 0: return from break.
 1: Eval another function.
 2: Try to figure another guess. [guess=1.00]
 3: Try to figure more. [count=5]
 4: Return to Top Level (an "abort" restart).
 5: Abort entirely from this (lisp) process.
[1c] CL-USER(201):
```

```
;;; invoke-restart を利用して平方根問題を出題する実行例

CL-USER(116): (let ((val 1))   ; val の平方根を求める関数 my-sqrt-func を定義する
               (defun my-sqrt-func (y)
                 (float (/ (+ y (/ val y)) 2)))
               (defun incf-sqrt-seed ()
                 (incf val)))
INCF-SQRT-SEED
CL-USER(117): (handler-bind ((simple-break
                             #'(lambda (cond)
                                 (declare (ignore cond))
                                 (format t "(sqrt ~,2F) = ?  " (incf-sqrt-seed))
                                 (invoke-restart 'change-function 'my-sqrt-func))))
               (find-fixed-point 'my-sqrt-func 1.0 0.00001 1000))
The fixed point of MY-SQRT-FUNC is 1.000000.
  [guess=1.00, count/limit=0/1000, sub/tolerance=0.000000/0.000010]
(sqrt 2.00) = ?  Enter a new guess: 1.4
The fixed point of MY-SQRT-FUNC is 1.414214.
  [guess=1.40, count/limit=3/1000, sub/tolerance=0.000000/0.000010]
(sqrt 3.00) = ?  Enter a new guess: 1.7
The fixed point of MY-SQRT-FUNC is 1.732051.
  [guess=1.70, count/limit=3/1000, sub/tolerance=0.000000/0.000010]
(sqrt 4.00) = ?  Enter a new guess: 2
The fixed point of MY-SQRT-FUNC is 2.000000.
  [guess=2.00, count/limit=1/1000, sub/tolerance=0.000000/0.000010]
(sqrt 5.00) = ?  Enter a new guess: 2.2
The fixed point of MY-SQRT-FUNC is 2.236068.
  [guess=2.20, count/limit=3/1000, sub/tolerance=0.000000/0.000010]
(sqrt 6.00) = ?  Enter a new guess:
```

3.5 Restart の実行

デバッガの中から :continiue <番号> を指定して対話的に restart する方法はこれまでの例で何度も使ってきました。ここでは、関数を利用した restart の方法について述べます。これにより、非対話的に restart することもできるようになります。

Restart を実行するには、invoke-restart, invoke-restart-interactively 関数を使います。前者は非対話的な restart 方法で、後者は対話的な restart 方法です。

これらの他、abort, muffle-warning, continue, use-value, store-value 関数も定義されています。

3.5.1 invoke-restart

記述例:

```
(invoke-restart <restart 名> [<arguments>])
```

今現在設定下にある restart を「active な restart」と呼びます。invoke-restart は、active な restart の中で restart 名 <restart 名> により指示された処理にジャンプします。この時、 restart 処理の引数には <arguments> が渡ります。

以下は 3.4.1 章の例を非対話的に restart した例です。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; error の場合は invoke-restart によりチェック範囲を修正する例
;;;
(defun change-range-value (val)
  (handler-bind ((error #'(lambda (cond)
                            (format t "I will set new limit.~%")
                            ;; 範囲外の場合は範囲を変更する
                            (invoke-restart 'set-new-limit (- val 10) (+ val 10)))))
    (check-value-with-restart val)))

CL-USER(131): (change-range-value -6)
I will set new limit.
-6    ; チェック値
-16   ; 変更した下限
4     ; 変更した上限
CL-USER(132):
```

3.4.1 章の my-sqrt-loop の例は無限 loop であり、対話的に値を取得することで処理が中断されます。この関数を invoke-restart で restart させると、中断せずに無限に loop することになります。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;
CL-USER(23): (handler-bind ((simple-break
                             #'(lambda (cond)
                                 (declare (ignore cond))
                                 ;; 前回の値に 1 足す restart を実行する
                                 (invoke-restart 'add-1))))
               (my-sqrt-loop))
sqrt    0 = 0.000000
sqrt    1 = 1.000000
sqrt    2 = 1.414214
sqrt    3 = 1.732051
sqrt    4 = 2.000000
sqrt    5 = 2.236068
sqrt    6 = 2.449490
sqrt    7 = 2.645751
sqrt    8 = 2.828427
...
sqrt  504 = 22.449944
sqrt  504 = 22.449944
...
```

3.5.2 invoke-restart-interactively

記述例:

```
(invoke-restart-interactively <restart 名>)
```

Active な restart の中で restart 名 <restart 名> で指示された処理にジャンプします。この時、restart 処理の引数は対話的に取得します。

以下は 3.4.1 章の my-sqrt-loop の例を対話的に restart した例です。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 対話的に数値を入力させ、無限に平方根を求め続ける例
;;;
CL-USER(81): (handler-bind ((simple-break
                             #'(lambda (cond)
                                 (declare (ignore cond))
                                 ;; restart 処理にジャンプするためデバッガは起動されない
                                 ;; 値の再設定を繰返す interface になる
                                 (invoke-restart-interactively 'store-value))))
               (my-sqrt-loop))
sqrt    0 = 0.000000
Enter a new number: 10
sqrt   10 = 3.162278
Enter a new number: 20
sqrt   20 = 4.472136
Enter a new number: 5
sqrt    5 = 2.236068
Enter a new number:
```

3.5.3 今現在 active な restart の取得 (compute-restarts, find-restart)

Restart 実行時、今現在 active でない (設定下にない) <restart 名>を指定すると control-error クラスのエラーが発生します。今現在 active な restart は compute-restarts 関数で取得することができます。また、今現在 active な restart を <restart 名> を指定して取得するには find-restart 関数を使います。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 今現在 active な restart のリストを表示する
;;;
CL-USER(415): (handler-bind ((error #'(lambda (cond)
                                        (print (compute-restarts))
                                        (abort))))
                (check-value-with-restart -1))

(#<restart STORE-VALUE for condition #<SIMPLE-ERROR @ #x40c25dfa> @ #x40c26cb2>
 #<restart SET-NEW-LIMIT for condition #<SIMPLE-ERROR @ #x40c25dfa> @ #x40c27b1a>
 #<restart ABORT @ #x40c24eea> #<restart ABORT @ #x402dcff2>)
CL-USER(416):
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 今現在 active な restart であるか確認し、active であれば restart を実行する
;;;
CL-USER(144): (handler-bind ((error #'(lambda (cond)
                                        (let ((r (find-restart 'set-new-limit)))
                                          (when r
                                            (format t "Restart ~S is named ~S~%" r (restart-name r))
                                            (format t "I will set a new limit.~%")
                                            (invoke-restart r -10 10))))))
                (check-value-with-restart -1))
Restart #<restart SET-NEW-LIMIT for condition
          #<SIMPLE-ERROR @ #x20b5580a>
          @ #x20b5752a> is named SET-NEW-LIMIT
I will set a new limit.
-1    ; チェック値
-10   ; 変更した下限
10    ; 変更した上限
CL-USER(145):

;;; top レベルでは set-new-limit の restart 設定下にないため、restart は取得できない

CL-USER(145): (find-restart 'set-new-limit)
NIL
```

3.5.4 定義済み restart 関数 (abort, muffle-warning, continue, use-value, store-value)

あらかじめ次の restart が定義されています。

```
;;; command top level に戻る
(abort)             ≡ (invoke-restart 'abort)

;;; warn 関数を終了させ処理を続ける
(muffle-warning)    ≡ (invoke-restart 'muffle-warning)

;;; break, cerror を終了させ処理を続ける
(continue)          ≡ (let ((r (find-restart 'continue)))
                        (when r (invoke-restart r)))

;;; 処理の結果として一時的に x を利用する
(use-value x)       ≡ (let ((r (find-restart 'use-value)))
                        (when r (invoke-restart r x)))

;;; 値を x に設定しなおして処理を続ける
(store-value x)     ≡ (let ((r (find-restart 'store-value)))
                        (when r (invoke-restart r x)))
```

自分で restart を設定する場合にも、意味的に正しいのであれば、これらの<restart 名> を積極的に

使うようにしましょう[9]。紛らわしい <restart 名> を自分で設定するよりは、理解しやすいプログラムになります。

3.6 Assertion

これまで紹介した機能を用いた様々な assertion 機能も定義されています。ざっと例をあげます。詳細は Common Lisp のドキュメントを参照してください。

3.6.1 assert

assert 以降の処理では、指定した条件が満されていることを保障します。関数の引数チェックや、呼び出した関数の結果チェックなどに効果的です。条件が満たされていない場合、 correctable error [10]が signaling されます。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 引数 val1 val2 が real 以外ならエラーを通知する
;;;
(defun add-num (val1 val2)
  (assert (and (realp val1) (realp val2))     ; false の時デバッガが起動する
      (val1 val2)                              ; デバッガ内で再入力させる値 (省略可)
    "Value is not real: ~A or ~A." val1 val2) ; メッセージ (インタンス指定も可)(省略可)
  (+ val1 val2))

;;; デバッガが起動される時の例

CL-USER(247): (add-num 'a 2)
Error: Value is not real: A or 2.

Restart actions (select using :continue):
 0: retry assertion with new values for VAL1, VAL2.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1] CL-USER(248): :cont 0            ; 値の再入力を選択
VAL1 currently has a value of A.
Do you want to supply a new value? (y or n) y
Enter a form to be evaluated: 1
VAL2 currently has a value of 2.
Do you want to supply a new value? (y or n) y
Enter a form to be evaluated: 2
3
```

[9]3.4.1 章の my-sqrt-loop の例では store-value を使っています。

[10]restart により復旧可能な error のこと

```
;;; ハンドラーで restart を実行した例

CL-USER(271): (handler-bind ((error #'(lambda (cond)
                                        (continue))))  ; 処理を継続させる
                (add-num 'a 2))
VAL1 currently has a value of A.      ; 対話的に値を再設定可能
Do you want to supply a new value? (y or n) y
Enter a form to be evaluated: 1
VAL2 currently has a value of 2.
Do you want to supply a new value? (y or n) y
Enter a form to be evaluated: 2
3
CL-USER(272):
```

3.6.2 check-type

型のチェックをし、一致していない場合は type-error を signaling します。store-value で値を設定しなおして restart できます。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 引数 val1 が integer, val2 が 0 から 99 の integer 型でないならエラーを通知する
;;;
(defun add-num2 (val1 val2)
  (check-type val1 integer)
  (check-type val2 (integer 0 99) "of range 0 to 99")
  (+ val1 val2))

;;; デバッガが起動される時の例

CL-USER(17): (add-num2 100 100)
Error: the value of VAL2 is 100, which is not of range 0 to 99.
  [condition type: TYPE-ERROR]

Restart actions (select using :continue):
 0: supply a new value for VAL2.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1] CL-USER(18): :cont 0          ; 値を再設定して処理を継続
Type a form to be evaluated: 50
150

;;; ハンドラーで store-value restart を実行した例

CL-USER(20): (handler-bind ((error
                             #'(lambda (cond)
                                 (format t "Illegal value: ~A.~%" cond)
                                 ;; エラーの場合は値を 60 に変更
                                 (invoke-restart 'store-value 60))))
               (add-num2 'a 100))
Illegal value: the value of VAL1 is A, which is not of type INTEGER..
Illegal value: the value of VAL2 is 100, which is not of range 0 to 99..
120     ; (+ 60 60)
CL-USER(21):
```

3.6.3 ccase

選択肢以外の値の時、correctable error を signaling します。`store-value` で値を設定しなおして restart できます。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 引数で指定された symbol を数字に変換する関数の例（ccase を利用した例）
;;;
(defun decode (x)
  (ccase x
    ((i uno) 1)
    ((ii dos) 2)
    ((iii tres) 3)
    ((iv cuatro) 4)))

;;; 選択肢にある値を指定した場合

CL-USER(342): (decode 'uno)
1

;;; 選択肢にない値を指定した場合

CL-USER(343): (decode 'v)
Error: V fell through a CCASE form.  The valid cases were I, UNO, II, DOS, III,
       TRES, IV, and CUATRO.
  [condition type: CASE-FAILURE]

Restart actions (select using :continue):
 0: supply a new value for X.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1] CL-USER(344): :cont 0          ; 値を設定しなおして処理を継続
Type a form to be evaluated: 'iii
3
CL-USER(345):
```

3.6.4 ecase

選択肢以外の値の時 non correctable error [11] を signaling します。`ccase` とは異なり、restart 処理は設定されていません。

[11] `restart` による復旧が不可能な `error` のこと

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 引数で指定された symbol を数字に変換する関数の例（ecase を利用した例）
(defun decode-by-ecase (x)
  (ecase x
    ((i uno) 1)
    ((ii dos) 2)
    ((iii tres) 3)
    ((iv cuatro) 4)))

;;; 選択肢にない値を指定した場合

CL-USER(349): (decode-by-ecase 'v)
Error: V fell through a ECASE form.  The valid cases were I, UNO, II, DOS, III,
       TRES, IV, and CUATRO.
  [condition type: CASE-FAILURE]

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] CL-USER(350): :cont 0       ; ccase とは異なり、 error から復旧する選択肢はない
CL-USER(351):
```

3.6.5 ctypecase

値が選択肢以外の型の時、 correctable error を signaling します。store-value で、値を設定しなおして restart できます。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 引数 val を integer に変換する例
;;;
(defun to-integer (val)
  (ctypecase val
    (integer val)
    (symbol (symbol-value val))
    (string (read-from-string val))))

;;; float を指定した場合

CL-USER(376): (defparameter x 100)      ; x に 100 を設定しておく
100
CL-USER(377): (to-integer 1.3)

Error: 1.3 fell through a CTYPECASE form.  The valid cases were INTEGER,
       SYMBOL, and STRING.
  [condition type: CASE-FAILURE]

Restart actions (select using :continue):
 0: supply a new value for VAL.
 1: Return to Top Level (an "abort" restart).
 2: Abort entirely from this (lisp) process.
[1] CL-USER(378): :cont 0               ; 値を設定しなおして処理を継続
Type a form to be evaluated: 'x         ; 新しい値 x を設定
100
CL-USER(379):
```

3.6.6 etypecase

値が選択肢以外の型の時、non correctable error を signaling します。`ctypecase` とは異なり、restart 処理は設定されていません。

4　condition システムの活用

実際のアプリケーション開発において condition システムをどのように使うべきか、活用するうえでの注意点も含めて説明します。

4.1　設計におけるコツ

4.1.1　condition の分類の重要性

ハンドラーは通知される `condition` のクラスを指定して設定しますので、あるハンドラーは、指定されたクラスを継承したクラスの condition もハンドリングします。このため、`condition` クラスの階層構造は非常に重要です。アプリケーションにおいてエラーが発生した時の動作毎に、親となる condition クラスをきちんと分類することが大切です。

以下に分類例を示します。

発生後のアプリケーションの動作別クラス	
終了	`(define-condition my-abort-condition-mixin () ())`
再起動	`(define-condition my-restart-condition-mixin () ())`
継続 (warning 時)	`(define-condition my-continue-warning-mixin (warning) ())`

発生時のユーザ通知方法別クラス	
メール通知	`(define-condition my-mail-condition-mixin () ())`
画面出力による通知	`(define-condition my-output-condition-mixin () ())`

この他、自分のアプリケーションで定義した condition か区別できるよう、ベースとなる condition を用意しておいた方が良いでしょう。ここでは `my-system-condition` としました。

```
(define-condition my-system-condition (simple-condition)
  ())
```

これらを親クラスとする子クラスを定義します。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; 子クラス定義例

;;; 発生後は停止し、ユーザには画面出力により通知する condition

(define-condition my-condition-a (my-output-condition-mixin
                                  my-abort-condition-mixin
                                  my-system-condition
                                  error)
  ())

;;; 発生後は再起動し、ユーザには画面出力により通知する condition

(define-condition my-condition-b (my-output-condition-mixin
                                  my-restart-condition-mixin
                                  my-system-condition
                                  error)
  ())

;;; 発生後は再起動し、ユーザにはメールにより通知する condition

(define-condition my-condition-c (my-mail-condition-mixin
                                  my-restart-condition-mixin
                                  my-system-condition
                                  error)
  ())
```

4.1.2 ハンドリング場所の重要性

ハンドリングの場所は、関数レベル、モジュールレベル、アプリケーションレベルで考えることができます。4.1.1 章で定義したような、停止再起動といった、アプリケーションの動作を左右する condition に対するハンドリングは、アプリケーションレベルでハンドリングすると使いやすいです。アプリケーションの停止や再起動といった処理を行なうハンドラーは、アプリケーションを自律的に動かすためデバッガの起動を抑止させる目的で書くことが多く、プログラムの深い位置にあると開発中は非常に邪魔な存在になります。このハンドリング設定を簡単にコントロールするためにも、アプリケーションのできるだけ外側にある方が便利です。

以下に前章の condition を利用した例をあげます。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; アプリケーションのメイン関数
;;;     開発時はメイン関数を直接呼んでアプリケーションを起動する。
;;;
(defun main ()
  ;; 乱数を発生させ、その値により様々なエラーを発生させる
  (let ((val (random 50)))
    (etypecase val
      ((integer  1  2) (error (make-condition 'my-condition-a
                               :format-control "The random value was ~D."
                               :format-arguments (list val))))
      ((integer  3 25) (error (make-condition 'my-condition-b
                               :format-control "The random value was ~D."
                               :format-arguments (list val))))
      ((integer 26 49) (error (make-condition 'my-condition-c
                               :format-control "The random value was ~D."
                               :format-arguments (list val)))))))
```

```
;;; アプリケーションの動作を左右するハンドラーはメインの外に書く

CL-USER(4): (tagbody
             main-tag
              (handler-bind ((my-output-condition-mixin
                               #'(lambda (cond)
                                   (format t "Display message & Restart:~28T~A~%" cond)))
                              (my-mail-condition-mixin
                               #'(lambda (cond)
                                   (format t "Send e-mail & Restart:~28T~A~%" cond)))
                              (my-continue-warning-mixin
                               #'(lambda (cond)
                                   (format t "Continue now...~%")
                                   (muffle-warning)))
                              (my-restart-condition-mixin
                               #'(lambda (cond)
                                   (format t "Restart now...~%")
                                   (go main-tag)))
                              (my-abort-condition-mixin
                               #'(lambda (cond)
                                   (format t "Abort now...~%")
                                   (exit 1)))
                              (error
                               #'(lambda (cond)
                                   (format t "Unknown Error: ~A...~%" cond)
                                   (go main-tag))))
                (main)))
Display message & Restart:  The random value was 4.
Restart now...
Display message & Restart:  The random value was 5.
Restart now...
Send e-mail & Restart:      The random value was 33.
Restart now...
Send e-mail & Restart:      The random value was 29.
Restart now...
Display message & Restart:  The random value was 3.
Restart now...
Send e-mail & Restart:      The random value was 40.
Restart now...
Display message & Restart:  The random value was 19.
Restart now...
Send e-mail & Restart:      The random value was 26.
Restart now...
Display message & Restart:  The random value was 14.
Restart now...
Send e-mail & Restart:      The random value was 41.
Restart now...
Unknown Error: 0 fell through a ETYPECASE form.  The valid cases were
               (INTEGER 1 2), (INTEGER 3 25), and (INTEGER 26 49)....
Display message & Restart:  The random value was 10.
Restart now...
Display message & Restart:  The random value was 2.
Abort now...
; killing "Run Bar Process"...
; killing "Editor Server"...
; killing "Connect to Emacs daemon"...
; killing "Initial Lisp Listener"...

----------------------------------------------------------------------
```

4.2 アプリケーションをデーモンプロセスとして動かすときには

4.2.1 スレッドが停止しないか注意

error 関数による signaling ではデバッガが起動され、対話的にコマンドを入力しなければその処理は停止したままになります。Allegro CL の場合は、その signaling があったスレッドのみ停止します。

停止することが許されないアプリケーションや、デーモンプロセスで起動するアプリケーションの場合は、各スレッドの一番外側で condition をハンドリングし、デバッガが起動されないようハンドラーを記述しておきます。(4.3 章も参照してください。)

4.2.2 zoom の重要性

アプリケーション開発において、エラーが発生した時の状況調査、つまりデバッグは非常に重要です。開発段階でのデバッグはもちろんのこと、リリース後、ユーザ使用時に起きたエラーは更に重要な意味を持ち、その情報を記録しておくことは重要です。

Allegro CL では、デバッガの中で :zoom コマンドを利用すると、エラーが発生した箇所と渡ってきた引数の値を見ることができます。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; zoom 出力を見る
;;;
(defun main (val)
  (func1 (+ val 1)))

(defun func1 (val)
  (func2 (+ val 2)))

(defun func2 (val)
  (error "Foo."))

CL-USER(464): (main 5)
Error: Foo.

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.
[1] CL-USER(465): :zoom
Evaluation stack:

   (ERROR "Foo.")
 ->(FUNC2 8)
   (FUNC1 6)
   (MAIN 5)
   (EVAL (MAIN 5))
   (TPL:TOP-LEVEL-READ-EVAL-PRINT-LOOP)
   (TPL:START-INTERACTIVE-TOP-LEVEL
      #<TERMINAL-SIMPLE-STREAM [initial terminal io] fd 0/1 @ #x40120882>
      #<Function TOP-LEVEL-READ-EVAL-PRINT-LOOP> ...)
[1] CL-USER(466):
```

開発中であれば開発者が :zoom コマンドや他のコマンドを入力して解析できますが、リリース後にデーモンで動作しているアプリケーションではそうはいきません。また、エラーが発生した状態でアプリケーションが停止しているのも良くありません。

Allegro CL では、handler-bind を利用した with-auto-zoom-and-exit マクロを提供しており、:zoom 出力をファイルに書き出すことが可能です。アプリケーションに合わせてこのマクロをカスタマイズして使いましょう。このマクロは以下にあります。

URL: http://franz.com/support/documentation/8.0/doc/other/with-auto-zoom-and-exit.htm
ファイル: *[Allegro CL install directory]*/src/autozoom.cl

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; with-auto-zoom-and-exit を利用した例
;;;
CL-USER(6): (with-auto-zoom-and-exit ("stack-trace.log")
              (main 5))
An unhandled error condition has been signalled: Foo.

; Exiting
----------------------------------------------------------------------
```

```
;;; stack-trace.log の内容（記述が長いため、一部省略しました）
CL-USER(2): (shell "cat stack-trace.log")
Evaluation stack:

 ->(SYS::..RUNTIME-OPERATION . :UNKNOWN-ARGS)
   (TPL:DO-COMMAND #1="zoom" :FROM-READ-EVAL-PRINT-LOOP NIL :COUNT T :ALL T)
       ........(省略).......
   (SIGNAL #22#)
   (ERROR #23="Foo.")
   [... EXCL::EVAL-AS-PROGN ]
   (BLOCK FUNC2 (ERROR #23#))
   [... EXCL::%EVAL ]
   (FUNC2 8)
   (SYS::..RUNTIME-OPERATION 8)
   [... EXCL::EVAL-AS-PROGN ]
   (BLOCK FUNC1 (FUNC2 (+ VAL 2)))
   [... EXCL::%EVAL ]
   (FUNC1 6)
   (SYS::..RUNTIME-OPERATION 6)
   [... EXCL::EVAL-AS-PROGN ]
   (BLOCK MAIN (FUNC1 (+ VAL 1)))
   [... EXCL::%EVAL ]
   (MAIN 5)
   (SYS::..RUNTIME-OPERATION 5)
   [... EXCL::EVAL-AS-PROGN ]
   (LET*
      ((#24=#:G103 (CONS 'ERROR #27=(LAMBDA (E) (WITH-STANDARD-IO-SYNTAX . #20#))))
       (#25=#:G101 (LIST #24#))
       (#26=#:G102 (CONS #25# (EXCL::FAST EXCL::*HANDLER-CLUSTERS*)))
       (EXCL::*HANDLER-CLUSTERS* #26#))
      (DECLARE (DYNAMIC-EXTENT #25# #26# #24# EXCL::*HANDLER-CLUSTERS*))
      . #28=((MAIN 5)))
       ........(省略).......
   (EXCL::START-LISP-EXECUTION T)
```

4.3 Thread 間のシグナリング

複数のスレッドで動いているアプリケーションにおいて、4.1.1 章で定義したようなアプリケーションの動作を左右する condition が各スレッドで発生する場合、そのハンドラーの記述方法 (ハンドラーの実行する場所) としては次の 2 通り考えられます。

各スレッドで処理	condition 発生時のアプリケーションの動作を記述したハンドラーマクロを定義し、全てのスレッドがそのマクロを利用する。 ハンドラー実行場所: 各スレッド 効果的な場面: 各スレッドが完全に独立していて、エラー発生時にそのスレッドのみ再起動する場合
メインスレッドで処理	各スレッドがメインスレッドにを割込みをかけて condition 通知し、メインスレッドで condition 発生時のアプケーションの動作をハンドリングする。 ハンドラー実行場所: メインスレッド 効果的な場面: エラー発生時にシステム全体を最初からやりなおす場合など

メインスレッドに割込みをかけるには、process-interrupt 関数を利用します。[12]次の例は上の 2 つの方法を合わせて利用した例です。

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; thread 内で処理すべき error 以外はメインスレッドに通知するマクロ定義
;;;
(define-condition thread-condition (simple-condition)
  ())

(defmacro with-my-application-handler ((main-process) &body body)
  `(tagbody
    thread-tag
     (handler-bind ((thread-condition  ; thread 内で対処すべきエラー
                     #'(lambda (cond)
                          ;; 自分のスレッドの処理を最初からやりなおす
                          (format t "Restart my thread...~%")
                          (go thread-tag)))
                    (error             ; メインスレッドで対処すべきエラー
                     #'(lambda (cond)
                          ;; メインスレッドに通知する
                          (mp:process-interrupt main-process
                                                #'(lambda () (error cond)))
                          (abort))))
       (progn ,@body))))
```

[12] Allegro CL では スレッドのことを Lisp process と呼んでいます

```
;;; 2 本のスレッドを起動した実行例

CL-USER(47): system:*current-process*   ; 現在の process（スレッド）を確認
#<MULTIPROCESSING:PROCESS Initial Lisp Listener(2) waiting for input @ #x402a2ce2>

;;;     process foo では my-condition-c を上げメインスレッドに通知
;;;     process bar では thread-condition を上げ、自分のスレッドを最初からやりなおす
CL-USER(48): (progn
               (mp:process-run-function "foo"
                 #'(lambda (main-process)
                     (with-my-application-handler (main-process)
                       (format t "In Foo.~%")
                       (error (make-condition 'my-condition-c))))
                 system:*current-process*)
               (mp:process-run-function "bar"
                 #'(lambda (main-process)
                     (with-my-application-handler (main-process)
                       (format t "In Bar.~%")
                       (sleep 1)
                       (error (make-condition 'thread-condition))))
                 system:*current-process*))
#<MULTIPROCESSING:PROCESS bar(9) @ #x7572d862>

;;; foo で発生した my-condition-c は main-process に通知。 main-process でデバッガが起動

CL-USER(49): In Foo.
In Bar.
Restart my thread...
In Bar.                                     ; bar は再起動を繰返している
Error: condition C:(mail & restart).        ; foo から上がったエラー
  [condition type: MY-CONDITION-C]

Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart).
 1: Abort entirely from this (lisp) process.

[1] CL-USER(50): system:*current-process*   ; 先程と同じ process で上がっていることがわかる
#<MULTIPROCESSING:PROCESS Initial Lisp Listener(2) waiting for input @ #x402a2ce2>

[1] CL-USER(51): :proc        ; foo は condition 通知後 abort している
P Bix   Dis   Sec    dSec Pri State    Process Name, Whostate, Arrest
*   2 29161 47197 47196.9   0 runnable Initial Lisp Listener, waiting for input
*   6  6012   173   173.3   0 waiting  *lisp-listener*, waiting for input
*   5   647     0     0.3   0 waiting  Editor Server, waiting for input
*   3    51     0     0.1   0 waiting  Connect to Emacs daemon, waiting for input
*   4     2     0     0.1   0 inactive Run Bar Process
*   7   125     0     0.0   0 waiting  Domain Name Server Client, waiting for input
*   9    14     0     0.0   0 waiting  bar, Sleeping
[1] CL-USER(52): :kill "bar"  ; bar を殺しておく
[1] CL-USER(53):
```

この例ように特定のスレッド (上の例では main-process) で一括にハンドリングを行なう場合、例えば main-process が、あるスレッドから割込みを受けてハンドラーを処理中に他スレッドから別の割込みがあった場合など、意図していない動作を引き起すことがあります。

一つのスレッドで一括してハンドリングを行なう場合は、割込みがネストしない、あるいはネストし

ても問題がないようなハンドラーのつくりにしておく必要があります。

5 最後に

Condition System を利用すれば、繁雑になりがちなダイナミックな処理の移動をシンプルに書くことができます。そして単なるアプリケーションにおける異常発生通知という場面だけでなく、次のような場面での対話的な機能も提供できるようになります。

- 対話的に設定ファイルを生成
  user 名, passwd, id などを、正しい値 (型) が入力されたか確認しながら、対話的に設定させます。

- Tree で構成される処理手順の分岐を対話的に選択
  これは実際に『広域 IP 網監視システム ENCORE[13]』で利用しています。

  ENCORE には経路障害の調査項目を表現した『ルール[14]』が定義されており、このルールを Tree 構成で表現した『発生し得る障害の仮説』が定義されています。

  ENCORE は経路の異常を検知するとオペレータに通知し、オペレータは仮説を起動します (ENCORE が自分で仮説を起動する場合もあります)。ある仮説が成り立った時、更に詳細な解析を行なうために次の仮説を起動しますが、この時、どの仮説を実行するかをオペレータに選択させる機能を備えています。オペレータはそこまでの仮説検証で得られた解析データをもとに、次の仮説を選びます。これを繰返すことで状況を絞り込み、最終的に異常の原因をつき止めます。この対話的に仮説を選択させる機能に Condition System を利用しています。

- プログラムで判断できないことを対話的に解決
  前項の ENCORE は、以前は対話機能はありませんでした。あらゆる検証を ENCORE 自身で行ない、その結果をまとめてオペレータに通知していました。仮説の数も多いため仮説検証に時間がかかる場合もあり、また、状況から既に把握済みの仮説検証が行なわれる場合もあり、無駄に時間を使っていました。これを回避するための対話機能の導入です。

  判断できないため発生し得る全ての仮説を実行していた所に、判断できないのであれば対話的に人間に判断させる機能を追加したわけです。

  これは ENCORE に限らず、他のアプリケーションでも同じ場面があるはずです。対話的に解決することで、アプリケーションの幅が必ず広がるはずです。更にはその解決手法をアプリケーションが学習していくことで、ユーザにとって使い勝手の良いアプリケーションになっていくはずです。

Condition System を利用して様々な対話インタフェースを考えることができ、そして簡単に実装することができます。また、例で示してきたように、Condition System をうまく使うと非常に幅広い用途に適した関数を書くことができます。そしてこれらを実現するための機能が ANSI 標準の言語仕様として提供されていることが Common Lisp の強みの一つになっています。

[13] An Inter-AS diagnostic ensemble system using cooperative reflector agents の略。複数の ISP 間にまたがる経路障害を自動的に検知し、原因解明を行なうシステム。

[14] 例えば、「`ping` は通るか」といったオペレータが実行する一つの調査項目のこと。

参考文献

[1] 『ANSI Common Lisp』
http://franz.com/support/documentation/8.0/ansicl/ansicl.htm

[2] Franz Inc., 『Allegro CL』
http://franz.com/

[3] Kent M. Pitman, 『Condition Handling in the Lisp Language Family』
http://www.nhplace.com/kent/Papers/Condition-Handling-2001.html

[4] Gerald Jay Sussman, Harold Abelson, Julie Sussman, 『計算機プログラムの構造と解釈』
ISBN4-89471-163-X (ピアソン, 2000 年)

[5] Franz Inc., 『with-auto-zoom-and-exit』
http://franz.com/support/documentation/8.0/doc/other/with-auto-zoom-and-exit.htm

[6] 明石修, 『マルチエージェントを用いた柔軟なインターネット運用アーキテクチャ』
http://jp.franz.com/base/seminar/2005-11-18/Akashi_pre.pdf