Exceptional Situations in Common Lisp

António Menezes Leitão

April 5, 2020

Exceptional Situations

Concepts

- Programs are written for *normal* situations:
  - Computing arithmetic operations over numbers
  - Retrieving items from lists
  - Reading data from files
  - Communicating with servers
- But there are also *exceptional* situations:
  - Computing arithmetic operations over non-numbers
  - Retrieving non-existent items from lists
  - Reading data from missing files
  - Communicating with servers when the network is down
- The distinction between *normal* and *exceptional* is arbitrary

Exceptional Situations

Concepts

- What happens when an exceptional situation is found?
  - It depends on the program
  - It depends on the programming language
- Tipically:
  - Return a distinguished value
  - Return an additional value
  - Set a variable
  - Call a function
  - Perform a special transfer of control
  - Stop the program and start the debugger
  - Abort the program
- We can only deal with exceptional situations that we can detect

Exceptional Situations

Concepts

- Some languages (e.g., Common Lisp), treat some exceptional situations as normal situations
- This might be good or bad

Common Lisp

```
> (sqrt -1)
```

Haskell

```
> sqrt (- 1)
```

Exceptional Situations

Concepts

- Some languages (e.g., Common Lisp), treat some exceptional situations as normal situations
- This might be good or bad

Common Lisp

```
> (sqrt -1)
#C(0.0 1.0)
```

Haskell

```
> sqrt (- 1)
NaN
```

Exceptional Situations

Concepts

- Some languages (e.g., Common Lisp), treat some exceptional situations as normal situations
- This might be good or bad

Common Lisp

```
> (sqrt -1)
#C(0.0 1.0)
> (asin 2)
```

Haskell

```
> sqrt (- 1)
NaN
> asin 2
```

Exceptional Situations

Concepts

- Some languages (e.g., Common Lisp), treat some exceptional situations as normal situations
- This might be good or bad

Common Lisp

```
> (sqrt -1)
#C(0.0 1.0)
> (asin 2)
#C(1.5707964 -1.316958)
```

Haskell

```
> sqrt (- 1)
NaN
> asin 2
NaN
```

Exceptional Situations

Concepts

- Some languages (e.g., Common Lisp), treat some exceptional situations as normal situations
- This might be good or bad

Common Lisp

```
> (sqrt -1)
#C(0.0 1.0)
> (asin 2)
#C(1.5707964 -1.316958)
> (first (list))
```

Haskell

```
> sqrt (- 1)
NaN
> asin 2
NaN
> head []
```

Exceptional Situations

Concepts

- Some languages (e.g., Common Lisp), treat some exceptional situations as normal situations
- This might be good or bad

Common Lisp

```
> (sqrt -1)
#C(0.0 1.0)
> (asin 2)
#C(1.5707964 -1.316958)
> (first (list))
NIL
```

Haskell

```
> sqrt (- 1)
NaN
> asin 2
NaN
> head []
*** Exception: head: empty list
```

Exceptional Situations

Concepts

- Some languages (e.g., Common Lisp), treat some exceptional situations as normal situations
- This might be good or bad

Common Lisp

```
> (sqrt -1)
#C(0.0 1.0)
> (asin 2)
#C(1.5707964 -1.316958)
> (first (list))
NIL
> (rest (list))
```

Haskell

```
> sqrt (- 1)
NaN
> asin 2
NaN
> head []
*** Exception: head: empty list
> tail []
```

Exceptional Situations

Concepts

- Some languages (e.g., Common Lisp), treat some exceptional situations as normal situations
- This might be good or bad

Common Lisp

```
> (sqrt -1)
#C(0.0 1.0)
> (asin 2)
#C(1.5707964 -1.316958)
> (first (list))
NIL
> (rest (list))
NIL
```

Haskell

```
> sqrt (- 1)
NaN
> asin 2
NaN
> head []
*** Exception: head: empty list
> tail []
*** Exception: tail: empty list
```

Exceptional Situations

Concepts

- Exceptional Situation: when there are several possible next steps and the program is unwilling or incapable of choosing among them
- Restarts: the possible next steps
- Signaling: asking for help
- Handlers: potential advisors that might help choosing the next step

Handlers

- Handlers might *decline* to help
- Handlers might choose a restart
- Handlers might take control of the computation

Exceptional Situations

Concepts

- Some exceptional situations are not errors
- They might be just unusual situations

Example: Print Lines

```
(defvar *line-end* 20)
```

Exceptional Situations

Concepts

- Some exceptional situations are not errors
- They might be just unusual situations

Example: Print Lines

```
(defvar *line-end* 20)

(defun print-line (str)
```

Exceptional Situations

Concepts

- Some exceptional situations are not errors
- They might be just unusual situations

Example: Print Lines

```
(defvar *line-end* 20)

(defun print-line (str)
  (let ((col 0))
```

Exceptional Situations

Concepts

- Some exceptional situations are not errors
- They might be just unusual situations

Example: Print Lines

```
(defvar *line-end* 20)

(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
```

Exceptional Situations

Concepts

- Some exceptional situations are not errors
- They might be just unusual situations

Example: Print Lines

```
(defvar *line-end* 20)

(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
```

Exceptional Situations

Concepts

- Some exceptional situations are not errors
- They might be just unusual situations

Example: Print Lines

```
(defvar *line-end* 20)

(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
```

Exceptional Situations

Concepts

- Some exceptional situations are not errors
- They might be just unusual situations

Example: Print Lines

```
(defvar *line-end* 20)

(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
```

Exceptional Situations

Concepts

- Some exceptional situations are not errors
- They might be just unusual situations

Example: Print Lines

```
(defvar *line-end* 20)

(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (signal 'line-end-limit)))
```

Exceptional Situations

Concepts

- Some exceptional situations are not errors
- They might be just unusual situations

Example: Print Lines

```
(defvar *line-end* 20)

(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (signal 'line-end-limit)))
    (write-char #\Newline)
```

Exceptional Situations

Concepts

- Some exceptional situations are not errors
- They might be just unusual situations

Example: Print Lines

```
(defvar *line-end* 20)

(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (signal 'line-end-limit)))
    (write-char #\Newline)
    col))
```

Exceptional Situations

Concepts

- Some exceptional situations are not errors
- They might be just unusual situations

Example: Print Lines

```
(defvar *line-end* 20)

(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (signal 'line-end-limit)))
    (write-char #\Newline)
    col))

(define-condition line-end-limit (condition) ())
```

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
```

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
```

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
14  ;;No problem
```

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
14  ;;No problem
> (print-line "Hi, everybody! How are you feeling today?")
```

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
14  ;;No problem
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How are you feeling today?
```

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
14  ;;No problem
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How are you feeling today?
20  ;;I signalled the situation but there was nobody listening
```

Signals

- If there are no handlers available, signals are ignored. . .

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
14  ;;No problem
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How are you feeling today?
20  ;;I signalled the situation but there was nobody listening
> (setq *break-on-signals* t)
```

Signals

- If there are no handlers available, signals are ignored. . .

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
14  ;;No problem
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How are you feeling today?
20  ;;I signalled the situation but there was nobody listening
> (setq *break-on-signals* t)
T
```

Signals

- If there are no handlers available, signals are ignored. . .

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
14  ;;No problem
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How are you feeling today?
20  ;;I signalled the situation but there was nobody listening
> (setq *break-on-signals* t)
T
> (print-line "Hi, everybody! How are you feeling today?")
```

Signals

- If there are no handlers available, signals are ignored. . .

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
14  ;;No problem
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How are you feeling today?
20  ;;I signalled the situation but there was nobody listening
> (setq *break-on-signals* t)
T
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
#<LINE-END-LIMIT @ #x716b7fd2>
break entered because of *break-on-signals*.
```

Signals

- If there are no handlers available, signals are ignored...
- ...unless `*break-on-signals*` is true

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
14  ;;No problem
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How are you feeling today?
20  ;;I signalled the situation but there was nobody listening
> (setq *break-on-signals* t)
T
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
#<LINE-END-LIMIT @ #x716b7fd2>
break entered because of *break-on-signals*.
```

Signals

- If there are no handlers available, signals are ignored...
- ...unless `*break-on-signals*` is true
- ...or the condition was signalled using `error`

Exceptional Situations

Handling Signals

- If there are handlers available, they are consulted from most specific to least specific

Example: Print Lines

```
(defun print-computing-excess (str)
  (let ((excess 0))
    (handler-bind ((line-end-limit
                    (lambda (condition)
                      (incf excess))))
      (print-line str))
    excess))
```

Exceptional Situations

Handling Signals

- If there are handlers available, they are consulted from most specific to least specific

Example: Print Lines

```
(defun print-computing-excess (str)
  (let ((excess 0))
    (handler-bind ((line-end-limit
                    (lambda (condition)
                      (incf excess))))
      (print-line str))
    excess))

> (print-computing-excess "Hi, everybody!")
```

Exceptional Situations

Handling Signals

- If there are handlers available, they are consulted from most specific to least specific

Example: Print Lines

```
(defun print-computing-excess (str)
  (let ((excess 0))
    (handler-bind ((line-end-limit
                    (lambda (condition)
                      (incf excess))))
      (print-line str))
    excess))

> (print-computing-excess "Hi, everybody!")
Hi, everybody!
```

Exceptional Situations

Handling Signals

- If there are handlers available, they are consulted from most specific to least specific

Example: Print Lines

```
(defun print-computing-excess (str)
  (let ((excess 0))
    (handler-bind ((line-end-limit
                    (lambda (condition)
                      (incf excess))))
      (print-line str))
    excess))

> (print-computing-excess "Hi, everybody!")
Hi, everybody!
0
```

Exceptional Situations

Handling Signals

- If there are handlers available, they are consulted from most specific to least specific

Example: Print Lines

```
(defun print-computing-excess (str)
  (let ((excess 0))
    (handler-bind ((line-end-limit
                    (lambda (condition)
                      (incf excess))))
      (print-line str))
    excess))

> (print-computing-excess "Hi, everybody!")
Hi, everybody!
0
> (print-computing-excess "Hi, everybody! How are you feeling today?")
```

Exceptional Situations

Handling Signals

- If there are handlers available, they are consulted from most specific to least specific

Example: Print Lines

```
(defun print-computing-excess (str)
  (let ((excess 0))
    (handler-bind ((line-end-limit
                    (lambda (condition)
                      (incf excess))))
      (print-line str))
    excess))
```

```
> (print-computing-excess "Hi, everybody!")
Hi, everybody!
0
> (print-computing-excess "Hi, everybody! How are you feeling today?")
Hi, everybody! How are you feeling today?
```

Exceptional Situations

Handling Signals

- If there are handlers available, they are consulted from most specific to least specific

Example: Print Lines

```
(defun print-computing-excess (str)
  (let ((excess 0))
    (handler-bind ((line-end-limit
                    (lambda (condition)
                      (incf excess))))
      (print-line str))
    excess))
```

```
> (print-computing-excess "Hi, everybody!")
Hi, everybody!
0
> (print-computing-excess "Hi, everybody! How are you feeling today?")
Hi, everybody! How are you feeling today?
21
```

Signaling a Condition/Requesting Handling

Print a string

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (signal 'line-end-limit)))
    (write-char #\Newline)
    col))
```

Signaling a Condition/Requesting Handling

Print a string

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (error 'line-end-limit)))
    (write-char #\Newline)
    col))
```

Signaling a Condition/Requesting Handling

Print a string

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (error 'line-end-limit)))
    (write-char #\Newline)
    col))
```

Result

```
> (print-line "Hi, everybody! How are you feeling today?")
```

Signaling a Condition/Requesting Handling

Print a string

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (error 'line-end-limit)))
    (write-char #\Newline)
    col))
```

Result

```
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
```

Signaling a Condition/Requesting Handling

Print a string

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (error 'line-end-limit)))
    (write-char #\Newline)
    col))
```

Result

```
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
Condition LINE-END-LIMIT was signalled.
```

Signaling a Condition/Requesting Handling

Print a string

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (error 'line-end-limit)))
    (write-char #\Newline)
    col))
```

Result

```
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
Condition LINE-END-LIMIT was signalled.
> (print-computing-excess "Hi, everybody! How are you feeling today?")
```

Signaling a Condition/Requesting Handling

Print a string

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (error 'line-end-limit)))
    (write-char #\Newline)
    col))
```

Result

```
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
Condition LINE-END-LIMIT was signalled.
> (print-computing-excess "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
```

Signaling a Condition/Requesting Handling

Print a string

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (error 'line-end-limit)))
    (write-char #\Newline)
    col))
```

Result

```
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
Condition LINE-END-LIMIT was signalled.
> (print-computing-excess "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
Condition LINE-END-LIMIT was signalled.
```

Handling a Condition

Print a string

```
(defun print-computing-excess (str)
  (let ((excess 0))
    (handler-bind ((line-end-limit
                    (lambda (condition)
                      (incf excess))))
      (print-line str))
    excess))
```

Aborting

- An handler can transfer control (possibly, aborting the current computation)

Handling a Condition/Aborting

Print a string

```
(defun print-maybe-aborting (str)
  (handler-bind ((line-end-limit
                  (lambda (condition)
                    (return-from print-maybe-aborting
                      "Line too long"))))
    (print-line str)))
```

Handling a Condition/Aborting

Print a string

```
(defun print-maybe-aborting (str)
  (handler-bind ((line-end-limit
                  (lambda (condition)
                    (return-from print-maybe-aborting
                      "Line too long"))))
    (print-line str)))
```

Result

Handling a Condition/Aborting

Print a string

```
(defun print-maybe-aborting (str)
  (handler-bind ((line-end-limit
                  (lambda (condition)
                    (return-from print-maybe-aborting
                      "Line too long"))))
    (print-line str)))
```

Result

```
> (print-maybe-aborting "Hi, everybody! How are you feeling today?")
```

Handling a Condition/Aborting

Print a string

```
(defun print-maybe-aborting (str)
  (handler-bind ((line-end-limit
                  (lambda (condition)
                    (return-from print-maybe-aborting
                      "Line too long"))))
    (print-line str)))
```

Result

```
> (print-maybe-aborting "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
```

Handling a Condition/Aborting

Print a string

```
(defun print-maybe-aborting (str)
  (handler-bind ((line-end-limit
                  (lambda (condition)
                    (return-from print-maybe-aborting
                      "Line too long"))))
    (print-line str)))
```

Result

```
> (print-maybe-aborting "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
"Line too long"
```

Handling a Condition/Aborting

Generalizing

```
(defun aborting-on-line-end-limit (f)
  (handler-bind ((line-end-limit
                  (lambda (condition)
                    (return-from aborting-on-line-end-limit
                      nil))))
    (funcall f)))
```

Handling a Condition/Aborting

Generalizing

```
(defun aborting-on-line-end-limit (f)
  (handler-bind ((line-end-limit
                  (lambda (condition)
                    (return-from aborting-on-line-end-limit
                      nil))))
    (funcall f)))
```

Result

Handling a Condition/Aborting

Generalizing

```
(defun aborting-on-line-end-limit (f)
  (handler-bind ((line-end-limit
                  (lambda (condition)
                    (return-from aborting-on-line-end-limit
                      nil))))
    (funcall f)))
```

Result

```
> (aborting-on-line-end-limit
    (lambda ()
      (print-line "Hi, everybody! How are you feeling today?")))
```

Handling a Condition/Aborting

Generalizing

```
(defun aborting-on-line-end-limit (f)
  (handler-bind ((line-end-limit
                  (lambda (condition)
                    (return-from aborting-on-line-end-limit
                      nil))))
    (funcall f)))
```

Result

```
> (aborting-on-line-end-limit
    (lambda ()
      (print-line "Hi, everybody! How are you feeling today?")))
Hi, everybody! How ar
```

Handling a Condition/Aborting

Generalizing

```
(defun aborting-on-line-end-limit (f)
  (handler-bind ((line-end-limit
                  (lambda (condition)
                    (return-from aborting-on-line-end-limit
                      nil))))
    (funcall f)))
```

Result

```
> (aborting-on-line-end-limit
    (lambda ()
      (print-line "Hi, everybody! How are you feeling today?")))
Hi, everybody! How ar
NIL
```

Handling a Condition/Propagating

Generalizing

```
(defun warning-on-signals (f)
  (handler-bind ((condition
                  (lambda (condition)
                    (warn "I saw a signal~%"))))
    (funcall f)))
```

Handling a Condition/Propagating

Generalizing

```
(defun warning-on-signals (f)
  (handler-bind ((condition
                  (lambda (condition)
                    (warn "I saw a signal~%"))))
    (funcall f)))
```

Result

Handling a Condition/Propagating

Generalizing

```
(defun warning-on-signals (f)
  (handler-bind ((condition
                  (lambda (condition)
                    (warn "I saw a signal~%"))))
    (funcall f)))
```

Result

```
> (aborting-on-line-end-limit
    (lambda ()
```

Handling a Condition/Propagating

Generalizing

```
(defun warning-on-signals (f)
  (handler-bind ((condition
                  (lambda (condition)
                    (warn "I saw a signal~%"))))
    (funcall f)))
```

Result

```
> (aborting-on-line-end-limit
    (lambda ()
      (warning-on-signals
        (lambda ()
```

Handling a Condition/Propagating

Generalizing

```
(defun warning-on-signals (f)
  (handler-bind ((condition
                  (lambda (condition)
                    (warn "I saw a signal~%"))))
    (funcall f)))
```

Result

```
> (aborting-on-line-end-limit
    (lambda ()
      (warning-on-signals
        (lambda ()
          (print-line "Hi, everybody! How are you feeling today?")))))
```

Handling a Condition/Propagating

Generalizing

```
(defun warning-on-signals (f)
  (handler-bind ((condition
                   (lambda (condition)
                     (warn "I saw a signal~%"))))
    (funcall f)))
```

Result

```
> (aborting-on-line-end-limit
    (lambda ()
      (warning-on-signals
        (lambda ()
          (print-line "Hi, everybody! How are you feeling today?")))))
Hi, everybody! How ar
```

Handling a Condition/Propagating

Generalizing

```
(defun warning-on-signals (f)
  (handler-bind ((condition
                   (lambda (condition)
                     (warn "I saw a signal~%"))))
    (funcall f)))
```

Result

```
> (aborting-on-line-end-limit
    (lambda ()
      (warning-on-signals
        (lambda ()
          (print-line "Hi, everybody! How are you feeling today?")))))
Hi, everybody! How ar
WARNING: I saw a signal
```

Handling a Condition/Propagating

Generalizing

```
(defun warning-on-signals (f)
  (handler-bind ((condition
                  (lambda (condition)
                    (warn "I saw a signal~%"))))
    (funcall f)))
```

Result

```
> (aborting-on-line-end-limit
    (lambda ()
      (warning-on-signals
        (lambda ()
          (print-line "Hi, everybody! How are you feeling today?")))))
Hi, everybody! How ar
WARNING: I saw a signal
NIL
```

Simplified Handling

The `handler-bind` form

```
(block handler
  (handler-bind ((condition-a (lambda (var-a)
                                (return-from handler expr-a)))
                 (condition-b (lambda (var-b)
                                (return-from handler expr-b)))
                 ...)
    expr))
```

Simplified Handling

The handler-bind form

```
(block handler
  (handler-bind ((condition-a (lambda (var-a)
                                (return-from handler expr-a)))
                 (condition-b (lambda (var-b)
                                (return-from handler expr-b)))
                 ...)
    expr))
```

The handler-case form

```
(handler-case expr
  (condition-a (var-a) expr-a)
  (condition-b (var-b) expr-b)
  ...)
```

Simplified Handling

The handler-bind form

```
(block handler
  (handler-bind ((condition-a (lambda (var-a)
                                (return-from handler expr-a)))
                 (condition-b (lambda (var-b)
                                (return-from handler expr-b)))
                 ...)
    expr))
```

The handler-case form

```
(handler-case expr
  (condition-a (var-a) expr-a)
  (condition-b (var-b) expr-b)
  ...)
```

Simplified Handling

The handler-bind form

```
(block handler
  (handler-bind ((condition-a (lambda (var-a)
                                (return-from handler expr-a)))
                 (condition-b (lambda (var-b)
                                (return-from handler expr-b)))
                 ...)
    expr))
```

The handler-case form

```
(handler-case expr
  (condition-a (var-a) expr-a)
  (condition-b (var-b) expr-b)
  ...)
```

Simplified Handling

The handler-bind form

```
(block handler
  (handler-bind ((condition-a (lambda (var-a)
                                (return-from handler expr-a)))
                 (condition-b (lambda (var-b)
                                (return-from handler expr-b)))
                 ...)
    expr))
```

The handler-case form

```
(handler-case expr
  (condition-a (var-a) expr-a)
  (condition-b (var-b) expr-b)
  ...)
```

Simplified Handling

The `handler-bind` form

```
(block handler
  (handler-bind ((condition-a (lambda (var-a)
                                (return-from handler expr-a)))
                 (condition-b (lambda (var-b)
                                (return-from handler expr-b)))
                 ...)
    expr))
```

The `handler-case` form

```
(handler-case expr
  (condition-a (var-a) expr-a)
  (condition-b (var-b) expr-b)
  ...)
```

Simplified Handling

The handler-bind form

```
(block handler
  (handler-bind ((condition-a (lambda (var-a)
                                (return-from handler expr-a)))
                 (condition-b (lambda (var-b)
                                (return-from handler expr-b)))
                 ...)
    expr))
```

The handler-case form

```
(handler-case expr
  (condition-a (var-a) expr-a)
  (condition-b (var-b) expr-b)
  ...)
```

Simplified Handling

The `handler-case` form

```
(handler-case expr
  (condition-a (var-a) expr-a ...)
  (condition-b (var-b) expr-b ...)
  ...)
```

Simplified Handling

The `handler-case` form

```
(handler-case expr
  (condition-a (var-a) expr-a ...)
  (condition-b (var-b) expr-b ...)
  ...)
```

The `try-catch` form

```
try {
  expr
} catch (condition_a var_a) {
  expr_a ...
} catch (condition_b var_b) {
  expr_b ...
}
```

Simplified Handling

The `handler-case` form

```
(handler-case expr
  (condition-a (var-a) expr-a ...)
  (condition-b (var-b) expr-b ...)
  ...)
```

The `try-catch` form

```
try {
  expr
} catch (condition_a var_a) {
  expr_a ...
} catch (condition_b var_b) {
  expr_b ...
}
```

Simplified Handling

The `handler-case` form

```
(handler-case expr
  (condition-a (var-a) expr-a ...)
  (condition-b (var-b) expr-b ...)
  ...)
```

The `try-catch` form

```
try {
  expr
} catch (condition_a var_a) {
  expr_a ...
} catch (condition_b var_b) {
  expr_b ...
}
```

Simplified Handling

The `handler-case` form

```
(handler-case expr
  (condition-a (var-a) expr-a ...)
  (condition-b (var-b) expr-b ...)
  ...)
```

The `try-catch` form

```
try {
  expr
} catch (condition_a var_a) {
  expr_a ...
} catch (condition_b var_b) {
  expr_b ...
}
```

Simplified Handling

The `handler-case` form

```
(handler-case expr
  (condition-a (var-a) expr-a ...)
  (condition-b (var-b) expr-b ...)
  ...)
```

The `try-catch` form

```
try {
  expr
} catch (condition_a var_a) {
  expr_a ...
} catch (condition_b var_b) {
  expr_b ...
}
```

Simplified Handling

The `handler-case` form

```
(handler-case expr
  (condition-a (var-a) expr-a ...)
  (condition-b (var-b) expr-b ...)
  ...)
```

The `try-catch` form

```
try {
  expr
} catch (condition_a var_a) {
  expr_a ...
} catch (condition_b var_b) {
  expr_b ...
}
```

Simplified Handling

The `unwind-protect` form

```
(unwind-protect expr
  cleanup
  ...)
```

Simplified Handling

The `unwind-protect` form

```
(unwind-protect expr
  cleanup
  ...)
```

The `try-finally` form

```
try {
  expr
} finaly {
  cleanup
  ...
}
```

Simplified Handling

The `unwind-protect` form

```
(unwind-protect expr
  cleanup
  ...)
```

The `try-finally` form

```
try {
  expr
} finaly {
  cleanup
  ...
}
```

Simplified Handling

The `unwind-protect` form

```
(unwind-protect expr
  cleanup
  ...)
```

The `try-finally` form

```
try {
  expr
} finaly {
  cleanup
  ...
}
```

Combined Handling

The `unwind-protect`/`handler-case` form

```
(unwind-protect
  (handler-case expr
    (condition-a expr-a ...)
    (condition-b expr-b ...)
    ...)
  cleanup ...)
```

Combined Handling

The `unwind-protect`/`handler-case` form

```
(unwind-protect
  (handler-case expr
    (condition-a expr-a ...)
    (condition-b expr-b ...)
    ...)
  cleanup ...)
```

The `try-catch-finally` form

```
try {
  expr
} catch (condition_a var_a) {
  expr_a ...
} catch (condition_b var_b) {
  expr_b ...
} finaly {
  cleanup ...
}
```

Restarts

Example: Print Lines

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (signal 'line-end-limit)))
    (write-char #\Newline)
    col))
```

Proceeding?

- A program signals an exceptional condition
- Because there might be more than one way to proceed
- It is the program's responsability to provide options
- But it is not its responsability to choose from them

Restarts

Example: Print Lines

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (restart-case (signal 'line-end-limit)
          (wrap ()
            (write-char #\Newline)
            (setf col 0))
          (truncate ()
            (return))
          (continue ()
            (incf col)))))
    (write-char #\Newline)
    col))
```

Signalling

- Use `signal` when the signal is ignorable

Restarts

Example: Print Lines

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (restart-case (signal 'line-end-limit)
          (wrap ()
            (write-char #\Newline)
            (setf col 0))
          (truncate ()
            (return))
          (continue ()
            (incf col)))))
    (write-char #\Newline)
    col))
```

Signalling

- Use `signal` when the signal is ignorable

Restarts

Example: Print Lines

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (restart-case (signal 'line-end-limit)
          (wrap ()
            (write-char #\Newline)
            (setf col 0))
          (truncate ()
            (return))
          (continue ()
            (incf col)))))
    (write-char #\Newline)
    col))
```

Signalling

- Use `signal` when the signal is ignorable

Restarts

Example: Print Lines

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (restart-case (signal 'line-end-limit)
          (wrap ()
            (write-char #\Newline)
            (setf col 0))
          (truncate ()
            (return))
          (continue ()
            (incf col)))))
    (write-char #\Newline)
    col))
```

Signalling

- Use `signal` when the signal is ignorable

Restarts

Example: Print Lines

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (restart-case (error 'line-end-limit) ;;We really need help
          (wrap ()
            (write-char #\Newline)
            (setf col 0))
          (truncate ()
            (return))
          (continue ()
            (incf col)))))
    (write-char #\Newline)
    col))
```

Signalling

- Use `error` when the signal is not ignorable

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
```

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
```

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
14  ;;No problem
```

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
14  ;;No problem
> (print-line "Hi, everybody! How are you feeling today?")
```

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
14  ;;No problem
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
#<LINE-END-LIMIT @ #x71831ea2>
  [Condition of type LINE-END-LIMIT]
```

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
14  ;;No problem
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
#<LINE-END-LIMIT @ #x71831ea2>
  [Condition of type LINE-END-LIMIT]

Restarts:
 0: [WRAP] WRAP
 1: [TRUNCATE] TRUNCATE
 2: [CONTINUE] CONTINUE
 3: [RETRY] Retry SLIME REPL evaluation request.
 4: [*ABORT] Return to SLIME's top level.
 5: [ABORT] Abort entirely from this (lisp) process.
```

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
14  ;;No problem
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
#<LINE-END-LIMIT @ #x71831ea2>
  [Condition of type LINE-END-LIMIT]

Restarts:
 0: [WRAP] WRAP
 1: [TRUNCATE] TRUNCATE
 2: [CONTINUE] CONTINUE
 3: [RETRY] Retry SLIME REPL evaluation request.
 4: [*ABORT] Return to SLIME's top level.
 5: [ABORT] Abort entirely from this (lisp) process.
```

Restarts

- We will improve restart's documentation later

Exceptional Situations

Example: Print Lines, Choosing WRAP

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (restart-case (error 'line-end-limit) ;;We need help
          (wrap ()
            (write-char #\Newline)
            (setf col 0))
          (truncate ()
            (return))
          (continue ()
            (incf col)))))
    (write-char #\Newline)
    col))
```

Exceptional Situations

Example: Print Lines, Choosing WRAP

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (restart-case (error 'line-end-limit) ;;We need help
          (wrap ()
            (write-char #\Newline)
            (setf col 0))
          (truncate ()
            (return))
          (continue ()
            (incf col)))))
    (write-char #\Newline)
    col))

> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
e you feeling today?
20
```

Exceptional Situations

Example: Print Lines, Choosing TRUNCATE

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (restart-case (error 'line-end-limit) ;;We need help
          (wrap ()
            (write-char #\Newline)
            (setf col 0))
          (truncate ()
            (return))
          (continue ()
            (incf col)))))
    (write-char #\Newline)
    col))
```

Exceptional Situations

Example: Print Lines, Choosing TRUNCATE

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (restart-case (error 'line-end-limit) ;;We need help
          (wrap ()
            (write-char #\Newline)
            (setf col 0))
          (truncate ()
            (return))
          (continue ()
            (incf col)))))
    (write-char #\Newline)
    col))

> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
20
```

Exceptional Situations

Example: Print Lines, Choosing `CONTINUE` twenty one times

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (restart-case (error 'line-end-limit) ;;We need help
          (wrap ()
            (write-char #\Newline)
            (setf col 0))
          (truncate ()
            (return))
          (continue ()
            (incf col)))))
    (write-char #\Newline)
    col))
```

Exceptional Situations

Example: Print Lines, Choosing `CONTINUE` twenty one times

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (restart-case (error 'line-end-limit) ;;We need help
          (wrap ()
            (write-char #\Newline)
            (setf col 0))
          (truncate ()
            (return))
          (continue ()
            (incf col)))))
    (write-char #\Newline)
    col))

> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How are you feeling today?
41
```

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
14  ;;No problem
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
#<LINE-END-LIMIT @ #x71831ea2>
  [Condition of type LINE-END-LIMIT]

Restarts:
 0: [WRAP] WRAP
 1: [TRUNCATE] TRUNCATE
 2: [CONTINUE] CONTINUE
 3: [RETRY] Retry SLIME REPL evaluation request.
 4: [*ABORT] Return to SLIME's top level.
 5: [ABORT] Abort entirely from this (lisp) process.
```

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
14  ;;No problem
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
#<LINE-END-LIMIT @ #x71831ea2>
  [Condition of type LINE-END-LIMIT]

Restarts:
 0: [WRAP] WRAP
 1: [TRUNCATE] TRUNCATE
 2: [CONTINUE] CONTINUE
 3: [RETRY] Retry SLIME REPL evaluation request.
 4: [*ABORT] Return to SLIME's top level.
 5: [ABORT] Abort entirely from this (lisp) process.
```

Restarts

- Let's improve the documentation

Exceptional Situations

Example: Print Lines

```
> (print-line "Hi, everybody!")
Hi, everybody!
14  ;;No problem
> (print-line "Hi, everybody! How are you feeling today?")
Hi, everybody! How ar
#<LINE-END-LIMIT @ #x71831ea2>
  [Condition of type LINE-END-LIMIT]

Restarts:
 0: [WRAP] Wrap the line
 1: [TRUNCATE] Truncate the line
 2: [CONTINUE] Keep printing the line
 3: [RETRY] Retry SLIME REPL evaluation request.
 4: [*ABORT] Return to SLIME's top level.
 5: [ABORT] Abort entirely from this (lisp) process.
```

Exceptional Situations

Example: Print Lines

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (restart-case (error 'line-end-limit)
          (wrap ()
            (write-char #\Newline)
            (setf col 0))
          (truncate ()
            (return))
          (continue ()
            (incf col)))))
    (write-char #\Newline)
    col))
```

Exceptional Situations

Example: Print Lines with Restart Documentation

```
(defun print-line (str)
  (let ((col 0))
    (dotimes (i (length str))
      (write-char (aref str i))
      (if (< col *line-end*)
        (incf col)
        (restart-case (error 'line-end-limit)
          (wrap () :report "Wrap the line"
            (write-char #\Newline)
            (setf col 0))
          (truncate () :report "Truncate the line"
            (return))
          (continue () :report "Keep printing the line"
            (incf col)))))
    (write-char #\Newline)
    col))
```

Exceptional Situations

Example: Print Lines in File

```
(defun print-lines-in-file (strs filename)
  (let ((f (open filename :direction :output :if-exists :supersede)))
    (let ((*standard-output* f))
      (dolist (str strs)









    (close f)))
```

Exceptional Situations

Example: Print Lines in File

```
(defun print-lines-in-file (strs filename)
  (let ((f (open filename :direction :output :if-exists :supersede)))
    (let ((*standard-output* f))
      (dolist (str strs)




        (print-line str))))
    (close f)))
```

Exceptional Situations

Example: Print Lines in File

```
(defun print-lines-in-file (strs filename)
  (let ((f (open filename :direction :output :if-exists :supersede)))
    (let ((*standard-output* f))
      (dolist (str strs)
        (handler-bind
            ((line-end-limit
               (lambda (condition)


          (print-line str))))
    (close f)))
```

Exceptional Situations

Example: Print Lines in File

```
(defun print-lines-in-file (strs filename)
  (let ((f (open filename :direction :output :if-exists :supersede)))
    (let ((*standard-output* f))
      (dolist (str strs)
        (handler-bind
            ((line-end-limit
               (lambda (condition)
                 (invoke-restart
                   (find-restart 'wrap condition)))))
          (print-line str))))
    (close f)))
```

Exceptional Situations

Example: Print Lines in File

```
(defun print-lines-in-file (strs filename)
  (let ((f (open filename :direction :output :if-exists :supersede)))
    (let ((*standard-output* f))
      (dolist (str strs)
        (handler-bind
            ((line-end-limit
               (lambda (condition)
                 (invoke-restart
                   (find-restart 'wrap condition)))))
          (print-line str))))
    (close f)))
```

Programmatic Selection of Restarts

- Separation between finding available restarts

Exceptional Situations

Example: Print Lines in File

```
(defun print-lines-in-file (strs filename)
  (let ((f (open filename :direction :output :if-exists :supersede)))
    (let ((*standard-output* f))
      (dolist (str strs)
        (handler-bind
            ((line-end-limit
               (lambda (condition)
                 (invoke-restart
                   (find-restart 'wrap condition)))))
          (print-line str))))
    (close f)))
```

Programmatic Selection of Restarts

- Separation between finding available restarts
- And invoking one of them

Exceptional Situations

Example: Print Lines in File

```
> (print-lines-in-file
    '("Hi, everybody! How are you feeling today?"
      "I hope you will enjoy what I'm about to say:"
      "Common Lisp is a great programming language!"
      "There, I said it.")
    "/tmp/text.txt")
```

Exceptional Situations

Example: Print Lines in File

```
> (print-lines-in-file
    '("Hi, everybody! How are you feeling today?"
      "I hope you will enjoy what I'm about to say:"
      "Common Lisp is a great programming language!"
      "There, I said it.")
    "/tmp/text.txt")
```

Result

```
Hi, everybody! How ar
e you feeling today?
I hope you will enjoy
 what I'm about to sa
y:
Common Lisp is a grea
t programming languag
e!
There, I said it.
```

Exceptional Situations

Programmatic Selection of Restarts

- Consider different severity levels

Example: Print Lines in File

```
(defun print-lines-in-file (strs filename)
  (let ((f (open filename :direction :output :if-exists :supersede)))
    (let ((*standard-output* f))
      (dolist (str strs)
        (handler-bind
            ((line-end-limit
                (lambda (condition)
                  (invoke-restart
                    (find-restart 'wrap condition)))))
          (print-line str))))
    (close f)))
```

Exceptional Situations

Programmatic Selection of Restarts

- Consider different severity levels

Example: Print Lines in File

```
(defun print-lines-in-file (strs filename)
  (let ((f (open filename :direction :output :if-exists :supersede)))
    (let ((*standard-output* f))
      (dolist (str strs)
        (handler-bind
            ((line-end-limit
                (lambda (condition)
                  (dolist (name '(wrap truncate continue
                                  abort explode nuclear-meltdown))
                    (let ((restart (find-restart name condition)))
                      (when restart
                        (invoke-restart restart))))))))
          (print-line str))))
    (close f)))
```

Exceptional Situations

Example: `data.txt`

```
John    Koenig  1234    44
Helena  Russel  4321    41
Alan    Carter  1Z34    30
Sandra  Benes   5678    30
Maya            9999    21
Bob     Mathias 1203    170
```

Exceptional Situations

Example: `data.txt`

```
John    Koenig  1234    44
Helena  Russel  4321    41
Alan    Carter  1Z34    30
Sandra  Benes   5678    30
Maya            9999    21
Bob     Mathias 1203    170
```

Validate Data

```
> (with-open-file (f "data.txt" :direction :input)
    (validate-stream f '(symbol symbol number (integer 0 99))))
```

Exceptional Situations

Example: `data.txt`

```
John    Koenig  1234    44
Helena  Russel  4321    41
Alan    Carter  1Z34    30
Sandra  Benes   5678    30
Maya            9999    21
Bob     Mathias 1203    170
```

Validate Data

```
> (with-open-file (f "data.txt" :direction :input)
    (validate-stream f '(symbol symbol number (integer 0 99))))

Wrong type in row 3 and col 3
   [Condition of type WRONG-TYPE-OF-COL]

Restarts:
 0: [RETRY] Retry SLIME REPL evaluation request.
 1: [*ABORT] Return to SLIME's top level.
 2: [ABORT] Abort entirely from this (lisp) process.
```

Exceptional Situations

Example: Validate Data

```
(defun validate-stream (stream types)
```

Exceptional Situations

Example: Validate Data

```
(defun validate-stream (stream types)
  (let ((length-types (length types)))
```

Exceptional Situations

Example: Validate Data

```
(defun validate-stream (stream types)
  (let ((length-types (length types)))
    (loop for line = (read-line stream nil nil)
          for row upfrom 1
```

Exceptional Situations

Example: Validate Data

```
(defun validate-stream (stream types)
  (let ((length-types (length types)))
    (loop for line = (read-line stream nil nil)
          for row upfrom 1
          while line
```

Exceptional Situations

Example: Validate Data

```
(defun validate-stream (stream types)
  (let ((length-types (length types)))
    (loop for line = (read-line stream nil nil)
          for row upfrom 1
          while line
          do (let ((fields (read-fields line)))
```

Exceptional Situations

Example: Validate Data

```
(defun validate-stream (stream types)
  (let ((length-types (length types)))
    (loop for line = (read-line stream nil nil)
          for row upfrom 1
          while line
          do (let ((fields (read-fields line)))
               (if (/= (length fields) length-types)
```

Exceptional Situations

Example: Validate Data

```
(defun validate-stream (stream types)
  (let ((length-types (length types)))
    (loop for line = (read-line stream nil nil)
          for row upfrom 1
          while line
          do (let ((fields (read-fields line)))
               (if (/= (length fields) length-types)
                 (error 'wrong-number-of-cols
                        :row row)
```

Exceptional Situations

Example: Validate Data

```
(defun validate-stream (stream types)
  (let ((length-types (length types)))
    (loop for line = (read-line stream nil nil)
          for row upfrom 1
          while line
          do (let ((fields (read-fields line)))
               (if (/= (length fields) length-types)
                 (error 'wrong-number-of-cols
                        :row row)
                 (loop for field in fields
                       for type in types
                       for col upfrom 1
```

Exceptional Situations

Example: Validate Data

```
(defun validate-stream (stream types)
  (let ((length-types (length types)))
    (loop for line = (read-line stream nil nil)
          for row upfrom 1
          while line
          do (let ((fields (read-fields line)))
               (if (/= (length fields) length-types)
                 (error 'wrong-number-of-cols
                        :row row)
                 (loop for field in fields
                       for type in types
                       for col upfrom 1
                       unless (typep field type)
```

Exceptional Situations

Example: Validate Data

```
(defun validate-stream (stream types)
  (let ((length-types (length types)))
    (loop for line = (read-line stream nil nil)
          for row upfrom 1
          while line
          do (let ((fields (read-fields line)))
               (if (/= (length fields) length-types)
                 (error 'wrong-number-of-cols
                        :row row)
                 (loop for field in fields
                       for type in types
                       for col upfrom 1
                       unless (typep field type)
                       do (error 'wrong-type-of-col
                                 :row row :col col)))))))
```

Exceptional Situations

Example: Validate Data

```
(defun validate-stream (stream types)
  (let ((length-types (length types)))
    (loop for line = (read-line stream nil nil)
          for row upfrom 1
          while line
          do (let ((fields (read-fields line)))
               (if (/= (length fields) length-types)
                 (error 'wrong-number-of-cols
                        :row row)
                 (loop for field in fields
                       for type in types
                       for col upfrom 1
                       unless (typep field type)
                       do (error 'wrong-type-of-col
                                 :row row :col col)))))))

(defun read-fields (str)
  (with-input-from-string (s str)
    (loop for field = (read s nil :eof)
          until (eq field :eof)
          collect field)))
```

Exceptional Situations

Validate Data: Conditions

```
(define-condition row-condition (condition)
  ((row :initarg :row :reader row-condition-row)))
```

Exceptional Situations

Validate Data: Conditions

```
(define-condition row-condition (condition)
  ((row :initarg :row :reader row-condition-row)))

(define-condition wrong-number-of-cols (row-condition)
  ()
  (:report (lambda (condition stream)
             (format stream "Incorrect number of cols in row ~A"
                     (row-condition-row condition)))))
```

Exceptional Situations

Validate Data: Conditions

```
(define-condition row-condition (condition)
  ((row :initarg :row :reader row-condition-row)))

(define-condition wrong-number-of-cols (row-condition)
  ()
  (:report (lambda (condition stream)
             (format stream "Incorrect number of cols in row ~A"
                     (row-condition-row condition)))))

(define-condition col-condition (row-condition)
  ((col :initarg :col :reader col-condition-col)))
```

Exceptional Situations

Validate Data: Conditions

```
(define-condition row-condition (condition)
  ((row :initarg :row :reader row-condition-row)))

(define-condition wrong-number-of-cols (row-condition)
  ()
  (:report (lambda (condition stream)
             (format stream "Incorrect number of cols in row ~A"
                     (row-condition-row condition)))))

(define-condition col-condition (row-condition)
  ((col :initarg :col :reader col-condition-col)))

(define-condition wrong-type-of-col (col-condition)
  ()
  (:report (lambda (condition stream)
             (format stream "Wrong type in row ~A and col ~A"
                     (row-condition-row condition)
                     (col-condition-col condition)))))
```

Exceptional Situations

Example: `data.txt`

```
John    Koenig  1234    44
Helena  Russel  4321    41
Alan    Carter  1Z34    30
Sandra  Benes   5678    30
Maya            9999    21
Bob     Mathias 1203    170
```

Exceptional Situations

Example: `data.txt`

```
John    Koenig  1234    44
Helena  Russel  4321    41
Alan    Carter  1Z34    30
Sandra  Benes   5678    30
Maya            9999    21
Bob     Mathias 1203    170
```

Validate Data

```
> (with-open-file (f "data.txt" :direction :input)
    (validate-stream f '(symbol symbol number (integer 0 99))))
```

Exceptional Situations

Example: `data.txt`

```
John    Koenig  1234    44
Helena  Russel  4321    41
Alan    Carter  1Z34    30
Sandra  Benes   5678    30
Maya            9999    21
Bob     Mathias 1203    170
```

Validate Data

```
> (with-open-file (f "data.txt" :direction :input)
    (validate-stream f '(symbol symbol number (integer 0 99))))

Wrong type in row 3 and col 3
   [Condition of type WRONG-TYPE-OF-COL]

Restarts:
 0: [RETRY] Retry SLIME REPL evaluation request.
 1: [*ABORT] Return to SLIME's top level.
 2: [ABORT] Abort entirely from this (lisp) process.
```

Exceptional Situations

Can we proceed?

```
(defun validate-stream (stream types)
  (let ((length-types (length types)))
    (loop for line = (read-line stream nil nil)
          for row upfrom 0
          while line
          do (let ((fields (read-fields line)))
               (if (/= (length fields) length-types)

                   (error 'wrong-number-of-cols
                          :row row)

                   (loop for field in fields
                         for type in types
                         for col upfrom 0
                         unless (typep field type)

                                (error 'wrong-type-of-col
                                       :row row :col col)
```

Exceptional Situations

Can we proceed?

```
(defun validate-stream (stream types)
  (let ((length-types (length types)))
    (loop for line = (read-line stream nil nil)
          for row upfrom 0
          while line
          do (let ((fields (read-fields line)))
               (if (/= (length fields) length-types)
                 (restart-case
                     (error 'wrong-number-of-cols
                            :row row)
                   (next-row () t))
                 (loop for field in fields
                       for type in types
                       for col upfrom 0
                       unless (typep field type)

                                (error 'wrong-type-of-col
                                       :row row :col col)
```

Exceptional Situations

Can we proceed?

```
(defun validate-stream (stream types)
  (let ((length-types (length types)))
    (loop for line = (read-line stream nil nil)
          for row upfrom 0
          while line
          do (let ((fields (read-fields line)))
               (if (/= (length fields) length-types)
                 (restart-case
                     (error 'wrong-number-of-cols
                            :row row)
                   (next-row () t))
                 (loop for field in fields
                       for type in types
                       for col upfrom 0
                       unless (typep field type)
                         do (restart-case
                                (error 'wrong-type-of-col
                                       :row row :col col)
                              (next-col () t)
                              (next-row () (return t)))))))))
```

Exceptional Situations

Example: `data.txt`

```
John    Koenig  1234    44
Helena  Russel  4321    41
Alan    Carter  1Z34    30
Sandra  Benes   5678    30
Maya            9999    21
Bob     Mathias 1203    170
```

Exceptional Situations

Example: `data.txt`

```
John    Koenig  1234    44
Helena  Russel  4321    41
Alan    Carter  1Z34    30
Sandra  Benes   5678    30
Maya            9999    21
Bob     Mathias 1203    170
```

Validate Data

```
> (with-open-file (f "data.txt" :direction :input)
    (validate-stream f '(symbol symbol number (integer 0 99))))
```

Exceptional Situations

Example: `data.txt`

```
John    Koenig  1234    44
Helena  Russel  4321    41
Alan    Carter  1Z34    30
Sandra  Benes   5678    30
Maya            9999    21
Bob     Mathias 1203    170
```

Validate Data

```
> (with-open-file (f "data.txt" :direction :input)
    (validate-stream f '(symbol symbol number (integer 0 99))))
Wrong type in row 3 and col 3
   [Condition of type WRONG-TYPE-OF-COL]

Restarts:
 0: [NEXT-COL] NEXT-COL
 1: [NEXT-ROW] NEXT-ROW
 2: [RETRY] Retry SLIME REPL evaluation request.
 3: [*ABORT] Return to SLIME's top level.
 4: [ABORT] Abort entirely from this (lisp) process.
```

Exceptional Situations

Example: `data.txt`

```
John    Koenig  1234    44
Helena  Russel  4321    41
Alan    Carter  1Z34    30
Sandra  Benes   5678    30
Maya            9999    21
Bob     Mathias 1203    170
```

Validate Data

```
> (with-open-file (f "data.txt" :direction :input)
    (validate-stream f '(symbol symbol number (integer 0 99))))
Wrong type in row 3 and col 3
   [Condition of type WRONG-TYPE-OF-COL]

Restarts:
 0: [NEXT-COL] NEXT-COL
 1: [NEXT-ROW] NEXT-ROW
 2: [RETRY] Retry SLIME REPL evaluation request.
 3: [*ABORT] Return to SLIME's top level.
 4: [ABORT] Abort entirely from this (lisp) process.
```

Exceptional Situations

Example: `data.txt`

```
John    Koenig  1234    44
Helena  Russel  4321    41
Alan    Carter  1Z34    30
Sandra  Benes   5678    30
Maya            9999    21
Bob     Mathias 1203    170
```

Validate Data

```
> (with-open-file (f "data.txt" :direction :input)
    (validate-stream f '(symbol symbol number (integer 0 99))))
Wrong type in row 3 and col 4
   [Condition of type WRONG-TYPE-OF-COL]

Restarts:
 0: [NEXT-COL] NEXT-COL
 1: [NEXT-ROW] NEXT-ROW
 2: [RETRY] Retry SLIME REPL evaluation request.
 3: [*ABORT] Return to SLIME's top level.
 4: [ABORT] Abort entirely from this (lisp) process.
```

Exceptional Situations

Example: `data.txt`

```
John    Koenig  1234    44
Helena  Russel  4321    41
Alan    Carter  1Z34    30
Sandra  Benes   5678    30
Maya            9999    21
Bob     Mathias 1203    170
```

Validate Data

```
> (with-open-file (f "data.txt" :direction :input)
    (validate-stream f '(symbol symbol number (integer 0 99))))
Wrong type in row 3 and col 4
   [Condition of type WRONG-TYPE-OF-COL]

Restarts:
 0: [NEXT-COL] NEXT-COL
 1: [NEXT-ROW] NEXT-ROW
 2: [RETRY] Retry SLIME REPL evaluation request.
 3: [*ABORT] Return to SLIME's top level.
 4: [ABORT] Abort entirely from this (lisp) process.
```

Exceptional Situations

Example: `data.txt`

```
John    Koenig  1234    44
Helena  Russel  4321    41
Alan    Carter  1Z34    30
Sandra  Benes   5678    30
Maya            9999    21
Bob     Mathias 1203    170
```

Validate Data

```
> (with-open-file (f "data.txt" :direction :input)
    (validate-stream f '(symbol symbol number (integer 0 99))))
Incorrect number of cols in row 5
   [Condition of type WRONG-NUMBER-OF-COLS]

Restarts:
 0: [NEXT-ROW] NEXT-ROW
 1: [RETRY] Retry SLIME REPL evaluation request.
 2: [*ABORT] Return to SLIME's top level.
 3: [ABORT] Abort entirely from this (lisp) process.
```

Exceptional Situations

Example: `data.txt`

```
John    Koenig  1234    44
Helena  Russel  4321    41
Alan    Carter  1Z34    30
Sandra  Benes   5678    30
Maya            9999    21
Bob     Mathias 1203    170
```

Validate Data

```
> (with-open-file (f "data.txt" :direction :input)
    (validate-stream f '(symbol symbol number (integer 0 99))))
Incorrect number of cols in row 5
   [Condition of type WRONG-NUMBER-OF-COLS]

Restarts:
 0: [NEXT-ROW] NEXT-ROW
 1: [RETRY] Retry SLIME REPL evaluation request.
 2: [*ABORT] Return to SLIME's top level.
 3: [ABORT] Abort entirely from this (lisp) process.
```

Exceptional Situations

Example: `data.txt`

```
John    Koenig  1234    44
Helena  Russel  4321    41
Alan    Carter  1Z34    30
Sandra  Benes   5678    30
Maya            9999    21
Bob     Mathias 1203    170
```

Validate Data

```
> (with-open-file (f "data.txt" :direction :input)
    (validate-stream f '(symbol symbol number (integer 0 99))))
Wrong type in row 6 and col 4
   [Condition of type WRONG-TYPE-OF-COL]

Restarts:
 0: [NEXT-COL] NEXT-COL
 1: [NEXT-ROW] NEXT-ROW
 2: [RETRY] Retry SLIME REPL evaluation request.
 3: [*ABORT] Return to SLIME's top level.
 4: [ABORT] Abort entirely from this (lisp) process.
```

Exceptional Situations

Programmatic Restart Invocation

- Known restart protocols allows automatic restart invocation
- Coordinate restart invocation with narrow conditions handling

Collecting All Errors

```
(defun collect-validation-errors (f)


      (funcall f))
```

Exceptional Situations

Programmatic Restart Invocation

- Known restart protocols allows automatic restart invocation
- Coordinate restart invocation with narrow conditions handling

Collecting All Errors

```
(defun collect-validation-errors (f)
  (let ((conditions (list)))
    (funcall f))
  (format t "~{~&~A~}" (reverse conditions))))
```

Exceptional Situations

Programmatic Restart Invocation

- Known restart protocols allows automatic restart invocation
- Coordinate restart invocation with narrow conditions handling

Collecting All Errors

```
(defun collect-validation-errors (f)
  (let ((conditions (list)))
    (handler-bind
```

```
      (funcall f))
    (format t "~{~&~A~}" (reverse conditions))))
```

Exceptional Situations

Programmatic Restart Invocation

- Known restart protocols allows automatic restart invocation
- Coordinate restart invocation with narrow conditions handling

Collecting All Errors

```
(defun collect-validation-errors (f)
  (let ((conditions (list)))
    (handler-bind
        ((row-condition




      (funcall f))
    (format t "~{~&~A~}" (reverse conditions))))
```

Exceptional Situations

Programmatic Restart Invocation

- Known restart protocols allows automatic restart invocation
- Coordinate restart invocation with narrow conditions handling

Collecting All Errors

```
(defun collect-validation-errors (f)
  (let ((conditions (list)))
    (handler-bind
        ((row-condition
            (lambda (condition)




      (funcall f))
    (format t "~{~&~A~}" (reverse conditions))))
```

Exceptional Situations

Programmatic Restart Invocation

- Known restart protocols allows automatic restart invocation
- Coordinate restart invocation with narrow conditions handling

Collecting All Errors

```
(defun collect-validation-errors (f)
  (let ((conditions (list)))
    (handler-bind
        ((row-condition
            (lambda (condition)
              (push condition conditions)


      (funcall f))
    (format t "~{~&~A~}" (reverse conditions))))
```

Exceptional Situations

Programmatic Restart Invocation

- Known restart protocols allows automatic restart invocation
- Coordinate restart invocation with narrow conditions handling

Collecting All Errors

```
(defun collect-validation-errors (f)
  (let ((conditions (list)))
    (handler-bind
        ((row-condition
            (lambda (condition)
              (push condition conditions)
              (dolist (name '(next-col next-row))
                (let ((restart (find-restart name condition)))
                  (when restart
                    (invoke-restart restart)))))))
      (funcall f))
    (format t "~{~&~A~}" (reverse conditions))))
```

Exceptional Situations

Validate Data

```
> (collect-validation-errors
    (lambda ()
      (with-open-file (f "data.txt" :direction :input)
        (validate-stream f '(symbol symbol number (integer 0 99))))))
```

Exceptional Situations

Validate Data

```
> (collect-validation-errors
    (lambda ()
      (with-open-file (f "data.txt" :direction :input)
        (validate-stream f '(symbol symbol number (integer 0 99))))))
Wrong type in row 3 and col 3
Wrong type in row 3 and col 4
Incorrect number of cols in row 5
Wrong type in row 6 and col 4
```

Communication with Restarts

- Condition signalling allows a function to pass information to another function
- The signalling function suspends execution
- Restart invocation allows the handling function to resume execution of the signalling function in some pre-established point


Kent M. Pitman.
Condition Handling in the Lisp Language Family, pages 39–59.
Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.


Peter Seibel.
Practical Common Lisp.
Apress, Berkeley, CA, USA, 2005.


Dorai Sitaram.
Handling control.
In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation: Albuquerque, New Mexico, June 23–25, 1993*, SIGPLAN notices; ISSN: 0362-1340; v. 28, no. 6 (June 1993), pages 147–155, New York, NY 10036, USA, 1993. ACM Press.


Guy L. Steele Jr.
Common Lisp: The Language.
Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, 1984.