# Control 2: Exceptions

Handout #42

CS251 Lecture 38

May 2, 2002

## Exception Handling

Want to be able to “signal” exceptional situations and handle them differently in different contexts.

Many languages provide exception systems:

- Java’s `throw` and `try/catch`
- ML’s `raise` and `handle`
- Common Lisp’s `throw` and `catch`

# Raise, handle, and trap

We will study exception handling in a version of Scheme extended with the following constructs:

- `(raise` *`T E`*`)`
  Evaluate *E* to value *V* and raise exception with tag *T* and value *V*.
- `(handle` *`T E_handler E_body`*`)` **; termination semantics**
  First evaluate *`E_handler`* to a one-argument handler function *`V_handler`*. Then evaluate *`E_body`* to value *`V_body`*. If no exception is encountered, return *`V_body`*. If an exception is raised with tag *`T`* and value *`V_val`*, the call to *handle* returns with the value of *`(V_handler V_val)`* evaluated at the point of the `handle`.
- `(trap` *`T E_handler E_body`*`)` **; resumption semantics**
  First evaluate *`E_handler`* to a one-argument handler function *`V_handler`*. Then evaluate *`E_body`* to value *`V_body`*. If no exception is encountered, return *`V_body`*. If an exception is raised with tag *T* and value *`V_val`*, the call to *raise* returns with the value of *`(V_handler V_val)`* evaluated at the point of the `raise`.

# Exception Handling Examples 1

```
(define test
  (lambda ()
    (let ((raiser (lambda (x)
                    (if (< x 0)
                        (raise negative x)
                        (if (even? x)
                            (raise even x)
                            x)))))
      (+ (raiser 1) (+ (raiser -3) (raiser 4))))))
```

What is the value of the following, where *`handler_1`* and *`handler_2`* range over {`handle`,`trap`}? First assume left-to-right argument evaluation, then right-to-left.

```
  (handler_1 negative (lambda (v) (- v))
    (handler_2 even (lambda (v) (* v v))
      (test)))
```

## Exception Handling Examples 2

What are the value of the following expressions, where *handler* ranges over {`handle`,`trap`}?

```
; Expression 1
(handler a (lambda (x) (+ 4000 x))
  (handler b (lambda (x) (+ 300 (raise a (+ x 4)))
    (handler a (lambda (x) (+ 20 x))
      (+ 1 (raise b 2)))))
```

```
; Expression 2
(handler c (lambda (x) (* x 10))
  (+ 1 (raise c (+ 2 (raise c 4)))))
```

## Exception Handling In ML

ML’s raise/handle uses **termination** semantics.

In `raise` *E*, *E* must evaluate to an exception packet created by an exception constructor (where exceptions are effectively an extensible datatype).

*E* `handle` *clauses* evaluates *E* and returns its value unless an exception is raised, in which case the matching clause in *clauses* is evaluated and its value is returned as the value of the `handle`.

## ML Exception Example

```
exception Neg of int
exception Even of int

fun raiser x =
    if x < 0 then
      raise (Neg x)
    else if (x mod 2) = 0 then
      raise (Even x)
    else
      x

fun test () = (raiser 1) + (raiser ~3) + (raiser 4)

fun innerTest () = test()
                     handle Neg(y) => raiser(7 + ~y)
                          | Even(z) => 3 * z

fun outerTest () = innerTest()
                     handle Neg(y) => ~y
                          | Even(z) => z * z
```

## Implementing `raise`

```
(raise tag E) desugars to  (raise-tag 'tag E)

(define raise-tag
  (lambda (tag value)
    (let ((handler
            ;; Look up handler in current handler env.
            ;; Handlers are dynamically scoped!
            (env-lookup tag (get-handler-env))))
      (if (unbound? handler)
          (error (string-append "Unhandled exception "
                                (symbol->string tag)
                                ": "))
          (handler value)))))
```

## Implementing `handle` and `trap` 1

```
(define with-handler
  (lambda (tag make-handler try-thunk)
    (begin
      (let ((old-env (get-handler-env)))
        (begin
          ;; Remember handler in dynamic environment
          (set-handler-env! (env-bind tag
                                      (make-handler old-env)
                                      (get-handler-env)))
          ;; Evaluate try-thunk
          (let ((try-value (try-thunk)))
            ;; In normal case, pop handler
            (begin
              (set-handler-env! old-env) ; reinstate old handler env.
              try-value))))))) ;; Return value
```

## Implementing `handle` and `trap` 2

```
(trap tag handler body) desugars to
  (let ((*handler* handler) ; only evaluate once
        (*thunk* (lambda () body))) ; avoid capturing *handler*
    (with-handler 'tag
      (lambda (old-env)
        (lambda (value) (*handler* value))) ; ignores old-env
      *thunk*))

(handle tag handler body) desugars to
  (let ((*handler* handler) ; only evaluate once
        (*thunk* (lambda () body))) ;avoid capturing *handler*
     (call-with-current-continuation
      (lambda (handle-cont)
        (with-handler 'tag
          (lambda (old-env)
            (lambda (value)
              ;; Invoking HANDLE-CONT returns directly to
              ;; appropriate handle, ignoring current continuation.
              (begin
                (set-handler-env! old-env) ; reinstall old-env
                (handle-cont (*handler* value)))))
          *thunk*))))
```