Part I

Implementing Errors

```
(test/exn (interp {+ 1 {1 1}})
          "not a function")
```

Change to

```
(test (interp {+ 1 {1 1}})
      (errorV "not a function"))
```

Implementing Errors

```
(define (continue k v)
  (type-case Cont k
    ...
    [(doAppK v-f next-k)
     (type-case Value v-f
       [closV ...]
       [else (errorV "not a function")])]))
```

Return **errorV** directly, dropping **k**

Implementing Errors

```
(define (lookup [n : Symbol] [env : Env] [k : Cont]) : Value
  (type-case (Listof Binding) env
    [empty (errorV "free variable")]
    [(cons b rst-env)
     (cond
       [(symbol=? n (bind-name b))
        (continue k (bind-val b))]
       [else (lookup n rst-env k)])]))
```

Implementing Errors

```
(define (num-op op l r k)
  (cond
   [(and (numV? l) (numV? r))
    (continue k (numV (op (numV-n l) (numV-n r))))]
   [else
    (errorV "not a number")]))

(define (num+ l r k)
  (num-op + l r k))
(define (num* l r k)
  (num-op * l r k))
```

Part 2

Catching Exceptions

```
(/ 1 0)
```

⇒ *division by zero*

Catching Exceptions

```
(try (/ 1 0)
     (lambda () +inf.0))
```

⇒ `+inf.0`

Catching Exceptions

```
(try (+ 1 0)
     (lambda () +inf.0))
```

⇒ 1

Catching Exceptions

```
(try (list 1 (/ 1 0) 3)
     (lambda () empty))
```

⇒ `empty`

Catching Exceptions

```
(cons 10
      (try (list 1 (/ 1 0) 3)
           (lambda () empty)))

⇒ (cons 10 empty)
```

Catching Exceptions

```
(try (try (list 1 (/ 1 0) 3)
          (lambda () empty))
     (lambda () (list 10)))
```

⇒ `empty`

Catching Exceptions

```
(try (try (list 1 (/ 1 0) 3)
          (lambda () (list (/ 1 0))))
     (lambda () (list 10)))

⇒ (list 10)
```

Language with `try`

```
<Exp>  ::=  <Number>
         |  <Symbol>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  {lambda {<Symbol>} <Exp>}
         |  {<Exp> <Exp>}
         |  {try <Exp> {lambda {} <Exp>}}
```

NEW

```
(test {try 0 {lambda {} 1}}
      (numV 0))
```

Language with `try`

```
<Exp>  ::=  <Number>
         |  <Symbol>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  {lambda {<Symbol>} <Exp>}
         |  {<Exp> <Exp>}
         |  {try <Exp> {lambda {} <Exp>}}
```

NEW

```
(test {try {0 0} {lambda {} 1}}
      (numV 1))
```

Language with `try`

```
<Exp>  ::=  <Number>
         |  <Symbol>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  {lambda {<Symbol>} <Exp>}
         |  {<Exp> <Exp>}
         |  {try <Exp> {lambda {} <Exp>}}
```

NEW

```
(test {+ {try 2 {lambda {} 1}}
         3}
      (numV 5))
```

Language with `try`

```
<Exp>  ::=  <Number>
         |  <Symbol>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  {lambda {<Symbol>} <Exp>}
         |  {<Exp> <Exp>}
         |  {try <Exp> {lambda {} <Exp>}}
```

NEW

```
(test {+ {try {2 2} {lambda {} 1}}
         3}
      (numV 4))
```

Language with `try`

```
<Exp>  ::=  <Number>
         |  <Symbol>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  {lambda {<Symbol>} <Exp>}
         |  {<Exp> <Exp>}
         |  {try <Exp> {lambda {} <Exp>}}
```

NEW

```
(test {try {try {0 0}
                {lambda {} 1}}
           {lambda {} 2}}
      (numV 1))
```

Language with `try`

```
<Exp>  ::=  <Number>
         |  <Symbol>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  {lambda {<Symbol>} <Exp>}
         |  {<Exp> <Exp>}
         |  {try <Exp> {lambda {} <Exp>}}
```

NEW

```
(test {try {try {0 0}
                {lambda {} {1 1}}}
           {lambda {} 2}}
      (numV 2))
```

Part 3

Expression and Parse

```
<Exp>  ::=  ...
        |   {try <Exp> {lambda {} <Exp>}}
```


```
(define-type Exp
  ....
  (tryE [body : Exp]
        [handle : Exp]))
```

```
(test (parse `{try {+ 1 2} {lambda {} 8}})
      (tryE (addE (numE 1) (numE 2))
            (numE 8)))
```

Interp

```
(define (interp a env k)
  (type-case Exp a
    ...
    [(tryE body handler)
     (interp body env (tryK handler env k))]))

(define (continue k v)
  (type-case Cont k
    ...
    [(tryK h env next-k)
     (continue next-k v)]))
```

Throwing Errors

Instead of just returning an **errorV**, look for a **tryK**:

Change

```
(errorV "not a number")
```

to

```
(escape k (errorV "not a number"))
```

Throwing Errors

Instead of just returning an **errorV**, look for a **tryK**:

```
(test (escape (doPlusK (numV 3)
                       (doneK))
              (errorV "fail"))
      (errorV "fail"))
```

Throwing Errors

Instead of just returning an **errorV**, look for a **tryK**:

```
(test (escape (doPlusK (numV 1)
                       (tryK (numE 2) mt-env
                             (doneK)))
              (errorV "fail"))
      (numV 2))
```

Throwing Errors

Instead of just returning an **errorV**, look for a **tryK**:

```
(test (escape (doPlusK (numV 1)
                       (tryK (numE 2) mt-env
                             (doPlusK (numV 3)
                                      (doneK))))
              (errorV "fail"))
      (numV 5))
```

Throwing Errors

Instead of just returning an **errorV**, look for a **tryK**:

```
(define (escape [k : Cont] [v : Value]) : Value
  (type-case Cont k
    [(doneK) v]
    [(plusSecondK r env next-k) (escape next-k v)]
    [(doPlusK v-l next-k) (escape next-k v)]
    [(multSecondK r env next-k) (escape next-k v)]
    [(doMultK v-l next-k) (escape next-k v)]
    [(appArgK a env next-k) (escape next-k v)]
    [(doAppK v-f next-k) (escape next-k v)]
    ...))
```

Throwing Errors

Instead of just returning an **`errorV`**, look for a **`tryK`**:

```
(define (escape [k : Cont] [v : Value]) : Value
  (type-case Cont k
    ...
    [(tryK h env next-k) (interp h env next-k)]))
```

Part 4

Continuation Jumps

The **`try`** form lets a programmer jump out to an enclosing context:

```
(+ 1
   (try (+ 2
           (+ 3
              (+ 4
                 (1 5))))
        (lambda () 0)))
```

jumps to

```
(+ 1 ●)
```

with code `0`

Continuation Jumps

The `let/cc` form lets a programmer jump out to any target context, and supply a value:

```
(+ 1
   (let/cc k1
     (+ 2
        (+ 3
           (let/cc k2
             (+ 4
                (k1 5)))))))
```

jumps to

```
(+ 1 ●)
```

with code `5`

Continuation Jumps

The `let/cc` form lets a programmer jump out to any target context, and supply a value:

```
(+ 1
   (let/cc k1
     (+ 2
        (+ 3
           (let/cc k2
             (+ 4
                (k2 5)))))))
```

jumps to

```
(+ 1 (+ 2 (+ 3 ●)))
```

with code `5`

Does it ever make sense to jump *in*?

Continuation Jumps

```
(define continue (lambda (n) n))

(let/cc esc
  (+ 1
     (+ 2
        (+ 3
           (+ 4
              (let/cc k
                (begin
                  (set! continue k)
                  (esc 0)))))))

(continue 5)
```

Continuation Jumps

```
(define continue (lambda (n) n))

(let/cc esc
  (+ 1
     (+ 2
        (+ 3
           (+ 4
              (let/cc k
                (begin
                  (set! continue k)
                  (esc 0)))))))

(continue 5)
```

```
(+ 1
   (+ 2
      (+ 3
         (+ 4
            ●))))
```

Continuation Jumps

```
(define continue (lambd

(let/cc esc
  (+ 1
     (+ 2
        (+ 3
           (+ 4
              (let/cc k
                (begin
                  (set! continue k)
                  (esc 0)))))))))

(continue 5)
```

```
(lambda (v)
  (+ 1
     (+ 2
        (+ 3
           (+ 4
              v)))))
```

Part 5

Language with `let/cc`

```
<Exp>  ::=  <Number>
         |  <Symbol>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  {lambda {<Symbol>} <Exp>}
         |  {<Exp> <Exp>}
         |  {let/cc <Symbol> <Exp>}
```

Implementing Continuations as Values

```
(define-type Value
  (numV [n : Number])
  (closV [arg : Symbol]
         [body : Exp]
         [env : Env])
  (contV [k : Cont]))
```

Implementing Continuations as Values

```
(define (interp a env k)
  (type-case Exp a
    ...
    [(let/ccE n body)
     (interp body
             (extend-env
              (bind n (contV k))
              env)
             k)]))
```

Implementing Continuations as Values

```
(define (continue k v)
  (type-case Cont k
    ...
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV n body c-env) ...]
       [(contV k-v) (continue k-v v)]
       [else (error ...)])]
    ...))
```

Part 6

Using Continuations

Few programs use **`let/cc`**...

Continuations are mostly useful for building other constructs:

- exception handling
- threads
- generators
- ...

Part 7

Generators

```
(define (make-numbers start-n)
  (generator
   yield ; <- binds for use below
   (local [(define (numbers n)
             (begin
               (yield n) ; <- yield a value
               (numbers (+ n 1))))]
     (numbers start-n))))

(define g (make-numbers 0))
(g) ; => 0
(g) ; => 1
(g) ; => 2
```

see **generator.rkt**

Part 8

Cooperative Threads

```
(define (count label n)
  (begin
    (pause) ; allows others to run
    (display label)
    (display (to-string n))
    (display "\n")
    (count label (+ n 1))))

(thread (lambda (vd) (count "a" 0)))
(thread (lambda (vd) (count "b" 0)))
(swap)
```

see **thread.rkt**