Part 1

Implementing Errors

```
check: interp(1 + 1(1))
       ~raises "not a function"
```

Change to

```
check: interp(1 + 1(1))
       ~is errorV("not a function")
```

Implementing Errors

```
fun continue(k, v):
  match k
  | ....
  | doAppK(v_f, next_k):
      match v_f
      | closV: ....
      | ~else: errorV("not a function")
```

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

Implementing Errors

```
fun lookup(n :: Symbol, env :: Env, k :: Cont) :: Value:
  match env
  | []: errorV("free variable")
  | cons(b, rst_env):
      cond
      | n == bind.name(b):
          continue(k, bind.val(b))
      | ~else: lookup(n, rst_env, k)
```

Implementing Errors

```
fun num_op(op, l, r, k):
  cond
  | (l, is_a, intV) && (r, is_a, intV):
      continue(k, intV(op(intV.n(l), intV.n(r))))
  | ~else:
      errorV("not a number")

fun num_plus(l, r, k):
  num_op(fun (x, y): x + y, l, r, k)
fun num_mult(l, r, k):
  num_op(fun (x, y):: x * y, l, r, k)
```

Part 2

Catching Exceptions

```
1 / 0
```

⟹ *division by zero*

Catching Exceptions

```
try:
  1 / 0
  ~catch: +inf.0

⇒ +inf.0
```

Catching Exceptions

```
try:
  1 + 0
  ~catch: +inf.0

⇒ 1
```

Catching Exceptions

```
try:
  [1, 1 / 0, 3]
  ~catch: empty

⇒ empty
```

Catching Exceptions

```
cons(10,
     try:
       [1, 1 / 0, 3]
       ~catch: empty)

⇒ cons(10, empty)
```

Catching Exceptions

```
try:
  try:
    [1, 1 / 0, 3]
    ~catch: empty
  ~catch: [10]

⇒ empty
```

Catching Exceptions

```
try:
  try:
    [1, 1 / 0, 3]
    ~catch: [1 / 0]
  ~catch: [10]

⇒ [10]
```

Language with try

```
<Exp> ::= <Int>
        | <Symbol>
        | <Exp> + <Exp>
        | <Exp> * <Exp>
        | fun (<Symbol>): <Exp>
        | <Exp>(<Exp>)
        | try: <Exp>
                ~catch: <Exp>
```

NEW

```
check: try:
         0
         ~catch: 1
       ~is intV(0)
```

Language with try

```
<Exp>  ::=  <Int>
         |  <Symbol>
         |  <Exp> + <Exp>
         |  <Exp> * <Exp>
         |  fun (<Symbol>): <Exp>
         |  <Exp>(<Exp>)
         |  try: <Exp>
                  ~catch: <Exp>
```

NEW

```
check:  try:
          0(0)
          ~catch: 1
        ~is intV(1)
```

Language with `try`

```
<Exp>  ::=  <Int>
          |  <Symbol>
          |  <Exp> + <Exp>
          |  <Exp> * <Exp>
          |  fun (<Symbol>): <Exp>
          |  <Exp>(<Exp>)
          |  try: <Exp>
                  ~catch: <Exp>
```

NEW

```
check: (try:
           2
           ~catch: 1)
         + 3
       ~is intV(5)
```

Language with `try`

```
<Exp>  ::=  <Int>
         |  <Symbol>
         |  <Exp> + <Exp>
         |  <Exp> * <Exp>
         |  fun (<Symbol>): <Exp>
         |  <Exp>(<Exp>)
         |  try: <Exp>
                  ~catch: <Exp>
```

NEW

```
check: (try:
          2(2)
          ~catch: 1)
         + 3
       ~is intV(4)
```

Language with `try`

```
<Exp>  ::=  <Int>
        |   <Symbol>
        |   <Exp> + <Exp>
        |   <Exp> * <Exp>
        |   fun (<Symbol>): <Exp>
        |   <Exp>(<Exp>)
        |   try: <Exp>
                  ~catch: <Exp>
```

NEW

```
check: try:
         try:
           0(0)
           ~catch: 1
         ~catch: 2
       ~is intV(1)
```

Language with `try`

```
<Exp>  ::=  <Int>
         |  <Symbol>
         |  <Exp> + <Exp>
         |  <Exp> * <Exp>
         |  fun (<Symbol>): <Exp>
         |  <Exp>(<Exp>)
         |  try: <Exp>
                 ~catch: <Exp>
```

NEW

```
check:  try:
          try:
            0(0)
            ~catch: 1(1)
          ~catch: 2
        ~is intV(2)
```

Part 3

Expression and Parse

```
<Exp>  ::=  ....
        |   try: <Exp>
                 ~catch: <Exp>
```

NEW

```
type Exp
....
| tryE(body :: Exp,
       handle :: Exp)
```

```
check: parse('try:
                1 + 2
                ~catch: 8')
       ~is tryE(addE(intE(1), intE(2)),
                intE(8))
```

Interp

```
fun interp(a, env, k):
  match a
  | ....
  | tryE(body, handler):
      interp(body, env, tryK(handler, env, k))

fun continue(k, v):
  match 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**:

```
check: escape(doPlusK(intV(3),
                      doneK()),
              errorV("fail"))
       ~is errorV("fail")
```

Throwing Errors

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

```
check: escape(doPlusK(intV(1),
                      tryK(intE(2), mt_env,
                           doneK())),
              errorV("fail"))
       ~is intV(2)
```

Throwing Errors

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

```
check: escape(doPlusK(intV(1),
                      tryK(intE(2), mt_env,
                           doPlusK(intV(3),
                                   doneK()))),
              errorV("fail"))
       ~is intV(5)
```

Throwing Errors

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

```
fun escape(k :: Cont, v :: Value) :: Value:
  match 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`:

```
fun escape(k :: Cont, v :: Value) :: Value:
  match 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(5)
       ~catch: 0)
```

jumps to

```
1 + ●
```

with value `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 + (let_cc k2:
              4 + k1(5)))
```

jumps to

```
1 + ●
```

with value 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 + (let_cc k2:
              4 + k2(5)))
```

jumps to

```
1 + 2 + ●
```

with value `5`

Does it ever make sense to jump *in*?

Continuation Jumps

```
def mutable continue = fun (n): n

let_cc esc:
  1 + 2 + 3 + 4 + (let_cc k:
                     block:
                       continue := k
                       esc(0))

continue(5)
```

Continuation Jumps

```
def mutable continue = fun (n): n

let_cc esc:
  1 + 2 + 3 + 4 + (let_cc k:
                     block:
                       continue := k
                       esc(0))

continue(5)
```

1 + 2 + 3 + 4 + ●

Continuation Jumps

```
def mutable continue = fun

let_cc esc:
  1 + 2 + 3 + 4 + (let_cc k:
                     block:
                       continue := k
                       esc(0))

continue(5)
```

```
fun (v):
  1 + 2 + 3 + 4 + v
```

Part 5

Language with `let_cc`

```
<Exp>  ::=  <Int>
        |   <Symbol>
        |   <Exp> + <Exp>
        |   <Exp> * <Exp>
        |   fun (<Symbol>) : <Exp>
        |   <Exp>(<Exp>)
        |   let_cc <Symbol>: <Exp>   NEW
```

Implementing Continuations as Values

```
type Value
| intV(n :: Int)
| closV(arg :: Symbol,
       body :: Exp,
       env :: Env)
| contV(k :: Cont)
```

Implementing Continuations as Values

```
fun interp(a, env, k):
  match a
  | ....
  | let_ccE(n, body):
      interp(body,
             extend_env(bind(n, contV(k)),
                        env),
             k)
```

Implementing Continuations as Values

```
fun continue(k, v):
  match k
  | ....
  | doAppK(v_f, next_k):
      match 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
- cooperative threads
- generators
- ....

Part 7

Generators

```
fun make_numbers(start_n):
  generator yield:
    block:
      fun numbers(n):
        begin:
          yield(n) // <- yield a value
          numbers(n + 1)
      numbers(start_n)

def g = make_numbers(0)
g() // => 0
g() // => 1
g() // => 2
```

see **`generator.rhm`**

Part 8

Cooperative Threads

```
fun count(label, n):
  block:
    pause() // allows others to run
    print(label)
    println(to_string(n))
    count(label, n + 1)

thread(fun (vd): count("a", 0))
thread(fun (vd): count("b", 0))
swap()
```

see **`thread.rhm`**