Continuations

The interpreter now consists of two main functions:

- **`eval-expression : expr env cont -> expval`**

exp= 1
env= {}
todo= [**done**]

- **`apply-cont : value cont -> expval`**

val= 1
todo= [**done**]

Control Constructs

Now that we've made control explicit, we can explore new control constructs:

- Exceptions
- Threads
- Programmer-accessible continuations

Exceptions

- Exceptions make error-handling easier and more reliable

```
FILE *f;
f = fopen("x.txt", "r");
fread(buffer, 1, 256, f);
/* oops: f might be NULL */
```

Exceptions

- Exceptions make error-handling easier and more reliable
- Exceptions can communicate "out of band" results more clearly

```
letrec index(n, l) = % find n in l, -1 if not there
        if null?(l)
         then -1
         else let p = (index n cdr(l))
                 in if zero?(add1(p))
                    then -1
                    else add1(p)
  in index(3, list(1, 2, 4))
```

Exceptions

- Exceptions make error-handling easier and more reliable
- Exceptions can communicate "out of band" results more clearly

```
letrec index(n, l) = % find n in l, exn if not there
        if null?(l)
         then raise -1
         else add1(index n cdr(l))
in try index(3, list(1, 2, 4))
   handle proc(x) -1
```

Exceptions

```
<expr>  ::=  ...
        ::=  try <expr> handle <expr>
        ::=  raise <expr>
```

raise 0 → *unhandled exception*

Exceptions

```
<expr>  ::=  ...
        ::=  try <expr> handle <expr>
        ::=  raise <expr>
```

raise sub1(1) → **raise** 0
→ *unhandled exception*

Exceptions

```
<expr>  ::=  ...
        ::=  try <expr> handle <expr>
        ::=  raise <expr>
```

try 10 **handle proc**(**x**)**x** → 10

Exceptions

<expr> ::= ...
::= **try** <expr> **handle** <expr>
::= **raise** <expr>

try 10 **handle let f** = **proc**(**x**)**x in f**
→ **try** 10 **handle proc**(**x**)**x**
→ 10

Exceptions

<expr> ::= ...
::= **try** <expr> **handle** <expr>
::= **raise** <expr>

try 10 **handle** 5 → *handler not a procedure*

Exceptions

<expr> ::= ...
::= **try** <expr> **handle** <expr>
::= **raise** <expr>

try raise 10 **handle proc**(**x**)**x** → **raise** 10
→ 10

- Where was the handler kept?

Evaluation with Exceptions: Raise

exp= raise 0
env= {}
todo= [**done**]

Evaluation with Exceptions: Raise

exp= **raise** 0
env= {}
todo= [**done**]

exp= 0
env= {}
todo= [**raise** [**done**]]

Evaluation with Exceptions: Raise

exp= 0
env= {}
todo= [**raise** [**done**]]

val= 0
todo= [**raise** [**done**]]

Evaluation with Exceptions: Raise

val= 0
todo= [**raise** [**done**]]

unhandled exception

Evaluation with Exceptions: Try

exp= **try** 10 **handle** **proc**(**x**)**x**
env= {}
todo= [**done**]

Evaluation with Exceptions: Try

exp= **try** 10 **handle** **proc**(**x**)**x**
env= {}
todo= [**done**]

exp= **proc**(**x**)**x**
env= {}
todo= [**try** 10 **in** {} [**done**]]

Evaluation with Exceptions: Try

exp= **proc**(**x**)**x**
env= {}
todo= [**try** 10 **in** {} [**done**]]

val= <**x**,**x**,{}>
todo= [**try** 10 **in** {} [**done**]]

Evaluation with Exceptions: Try

val= <**x**,**x**,{}>
todo= [**try** 10 **in** {} [**done**]]

exp= 10
env= {}
todo= [**handle** <**x**,**x**,{}> [**done**]]

Evaluation with Exceptions: Try

exp= 10
env= {}
todo= [**handle** <**x**,**x**,{}> [**done**]]

val= 10
todo= [**handle** <**x**,**x**,{}> [**done**]]

Evaluation with Exceptions: Try

val= 10
todo= [**handle** <**x**,**x**,{}> [**done**]]

val= 10
todo= [**done**]

Evaluation with Exceptions: Handle

exp= **try raise** 10 **handle proc**(**x**)**x**
env= {}
todo= [**done**]

Evaluation with Exceptions: Handle

exp= **try raise** 10 **handle proc**(**x**)**x**
env= {}
todo= [**done**]

exp= **proc**(**x**)**x**
env= {}
todo= [**try raise** 10 **in** {} [**done**]]

Evaluation with Exceptions: Handle

exp= **proc**(**x**)**x**
env= {}
todo= [**try raise** 10 **in** {} [**done**]]

val= <**x**,**x**,{}>
todo= [**try raise** 10 **in** {} [**done**]]

Evaluation with Exceptions: Handle

val= <**x**,**x**,{}>
todo= [**try raise** 10 **in** {} [**done**]]

exp= **raise** 10
env= {}
todo= [**handle** <**x**,**x**,{}> [**done**]]

Evaluation with Exceptions: Handle

exp= **raise** 10
env= {}
todo= [**handle** <**x**,**x**,{}> [**done**]]

exp= 10
env= {}
todo= [**raise** [**handle** <**x**,**x**,{}> [**done**]]]

Evaluation with Exceptions: Handle

exp= 10
env= {}
todo= [**raise** [**handle** <**x**,**x**,{}> [**done**]]]

val= 10
todo= [**raise** [**handle** <**x**,**x**,{}> [**done**]]]

Evaluation with Exceptions: Handle

val= 10
todo= [**raise** [**handle** <**x**,**x**,{}> [**done**]]]

exp= **x**
env= {**x**=10}
todo= [**done**]

Evaluation with Exceptions: Handle

exp= **x**
env= {**x**=10}
todo= [**done**]

val= 10
todo= [**done**]

Evaluation with Exceptions: In Context

exp= **sub1**(**try** **add1**(**add1**(**raise** 10)) **handle** **proc**(**x**)**x**)
env= {}
todo= [**done**]

Evaluation with Exceptions: In Context

exp= **sub1**(**try** **add1**(**add1**(**raise** 10)) **handle** **proc**(**x**)**x**)
env= {}
todo= [**done**]

exp= **try** **add1**(**add1**(**raise** 10)) **handle** **proc**(**x**)**x**
env= {}
todo= [-1 [**done**]]

Evaluation with Exceptions: In Context

exp= **try** **add1**(**add1**(**raise** 10)) **handle** **proc**(**x**)**x**
env= {}
todo= [-1 [**done**]]

exp= **proc**(**x**)**x**
env= {}
todo= [**try** **add1**(**add1**(**raise** 10)) **in** {} [-1 [**done**]]]

Evaluation with Exceptions: In Context

exp= **proc**(**x**)**x**
env= {}
todo= [**try** **add1**(**add1**(**raise** 10)) **in** {} [-1 [**done**]]]

val= <**x**,**x**,{}>
todo= [**try** **add1**(**add1**(**raise** 10)) **in** {} [-1 [**done**]]]

Evaluation with Exceptions: In Context

val= <**x**,**x**,{}>
todo= [**try** **add1**(**add1**(**raise** 10)) **in** {} [-1 [**done**]]]

exp= **add1**(**add1**(**raise** 10))
env= {}
todo= [**handle** <**x**,**x**,{}> [-1 [**done**]]]

Evaluation with Exceptions: In Context

exp= **add1**(**add1**(**raise** 10))
env= {}
todo= [**handle** <**x**,**x**,{}> [-1 [**done**]]]

exp= **add1**(**raise** 10)
env= {}
todo= [+1 [**handle** <**x**,**x**,{}> [-1 [**done**]]]]

Evaluation with Exceptions: In Context

exp= **add1**(**raise** 10)
env= {}
todo= [+1 [**handle** <**x**,**x**,{}> [-1 [**done**]]]]

exp= **raise** 10
env= {}
todo= [+1 [+1 [**handle** <**x**,**x**,{}> [-1 [**done**]]]]]

Evaluation with Exceptions: In Context

exp= **raise** 10
env= {}
todo= [+1 [+1 [**handle** <**x**,**x**,{}> [-1 [**done**]]]]]

exp= 10
env= {}
todo= [**raise** [+1 [+1 [**handle** <**x**,**x**,{}> [-1 [**done**]]]]]]

Evaluation with Exceptions: In Context

exp= 10
env= {}
todo= [**raise** [+1 [+1 [**handle** <**x**,**x**,{}> [-1 [**done**]]]]]]

val= 10
todo= [**raise** [+1 [+1 [**handle** <**x**,**x**,{}> [-1 [**done**]]]]]]

Evaluation with Exceptions: In Context

val= 10
todo= [**raise** [+1 [+1 [**handle** <**x**,**x**,{}> [-1 [**done**]]]]]]

exp= **x**
env= {**x**=10}
todo= [-1 [**done**]]

Evaluation with Exceptions: In Context

exp= **x**
env= {**x**=10}
todo= [-1 [**done**]]

val= 10
todo= [-1 [**done**]]

Evaluation with Exceptions: In Context

val= 10
todo= [-1 [**done**]]

val= 9
todo= [**done**]

Exceptions: Implementation

In **`eval-expression`**:

```
(try-exp (body-exp handler-exp)
  (eval-expression
   handler-exp env (try-cont body-exp env cont))
```

In **`apply-cont`**:

```
  (try-cont (body-exp env cont)
    (if (proc? val)
        (eval-expression
         body-exp env (handle-cont val cont))
        (error "handler not a proc"))
  (handle-cont (handler cont)
     (apply-cont cont val))
```

Exceptions: Implementation

In **`eval-expression`**:

```
        (raise-exp (expr)
           (eval-expression
            expr env (raise-cont cont)))
```

In **`apply-cont`**:

```
          (raise-cont (cont)
             (find-handler val cont))
```

Exceptions: Implementation

```
   (define (find-handler val cont)
     (cases continuation cont
       (handle-cont (handler cont)
          (apply-proc handler val cont))
       (done-cont ()
          (error "unhandled exception"))

       ; All others: look in the rest
       (prim-other-cont (prim arg2 env cont)
          (find-handler val cont))
       ....
```

Threads

- So far, our languages have been ***single-threaded***
- We can add a **spawn** form to our language to make it multithreaded

<expr>	::=	...
	::=	**spawn** <expr>
<prim>	::=	...
	::=	**print**

- Note: threads are only useful with side effects

Threads

- To implement threads, we need some way of packaging up all information about a computation
- The arguments to **`eval-expression`** and **`apply-cont`** *are* complete information!
- So, all we need is a way to keep a queue of thread states, and a way to switch to a different state occasionally

Thread Implementation

- Rename **`eval-expression`** to **`do-eval-expression`**
- Define a new **`eval-expression`**:

```
(define (eval-expression exp env cont)
  (if (time-to-swap!?)
      (swap-thread! 'eval (list exp env cont))
      (do-eval-expression exp env cont))))
```

Thread Implementation

- Rename **`apply-cont`** to **`do-apply-cont`**
- Define a new **`apply-cont`**:

```
(define (apply-cont cont val)
  (if (time-to-swap!?)
      (swap-thread! 'cont (list cont val))
      (do-apply-cont cont val)))
```

Thread Implementation

(implement the rest in DrScheme)

Continuations as Values

- With procedures, a program has a way to grab the current environment the restore it later
- What if we let programs grab and restore the environment?

First-class continuations

An expressed value is
- a number
- a proc
- a continuation

Continuations as Values

- With procedures, a program has a way to grab the current environment the restore it later
- What if we let programs grab and restore the environment?

First-class continuations

<expr> ::= ...
::= **letcc** <id> **in** <expr>
::= **continue** <expr> <expr>

Evaluation with letcc

exp= **letcc k** +(**continue k** 2, 1)
env= {}
todo= [**done**]

Evaluation with letcc

exp= **letcc k** +(**continue k** 2, 1)
env= {}
todo= [**done**]

exp= +(**continue k** 2, 1)
env= {**k**=[**done**]}
todo= [**done**]

Evaluation with letcc

exp= +(**continue k** 2, 1)
env= {**k**=[**done**]}
todo= [**done**]

exp= **continue k** 2
env= {**k**=[**done**]}
todo= [**addexp** 1 {**k**=[**done**]} [**done**]]

Evaluation with letcc

exp= **continue k** 2
env= {**k**=[**done**]}
todo= [**addexp** 1 {**k**=[**done**]} [**done**]]

exp= **k**
env= {**k**=[**done**]}
todo= [**contexp** 2 {**k**=[**done**]} [**addexp** 1 {**k**=[**done**]} [**done**]]]

Evaluation with letcc

exp= **k**
env= {**k**=[**done**]}
todo= [**contexp** 2 {**k**=[**done**]} [**addexp** 1 {**k**=[**done**]} [**done**]]]

val= [**done**]
todo= [**contexp** 2 {**k**=[**done**]} [**addexp** 1 {**k**=[**done**]} [**done**]]]

Evaluation with letcc

val= [**done**]
todo= [**contexp** 2 {**k**=[**done**]} [**addexp** 1 {**k**=[**done**]} [**done**]]]

exp= 2
env= {**k**=[**done**]}
todo= [**cont** [**done**] [**addexp** 1 {**k**=[**done**]} [**done**]]]

Evaluation with letcc

exp= 2
env= {**k**=[**done**]}
todo= [**cont** [**done**] [**addexp** 1 {**k**=[**done**]} [**done**]]]

val= 2
todo= [**cont** [**done**] [**addexp** 1 {**k**=[**done**]} [**done**]]]

Evaluation with letcc

val= 2
todo= [**cont** [**done**] [**addexp** 1 {**k**=[**done**]} [**done**]]]

val= 2
todo= [**done**]

Evaluation with letcc

The **cont** continuation never uses the rest

val= [**done**]
todo= [**contexp** 2 {**k**=[**done**]} [**addexp** 1 {**k**=[**done**]} [**done**]]]

Evaluation with letcc

The **cont** continuation never uses the rest

val= [**done**]
todo= [**contexp** 2 {**k**=[**done**]} [**addexp** 1 {**k**=[**done**]} [**done**]]]

exp= 2
env= {**k**=[**done**]}
todo= [**cont** [**done**]]

Evaluation with letcc

The **cont** continuation never uses the rest

exp= 2
env= {**k**=[**done**]}
todo= [**cont** [**done**]]

val= 2
todo= [**cont** [**done**]]

Evaluation with letcc

The **cont** continuation never uses the rest

val= 2
todo= [**cont** [**done**]]

val= 2
todo= [**done**]

Uses for letcc

First-class continuations are extremely powerful:

- **letcc** can be used instead of exceptions
- **letcc** can be used to re-play a computation
- **letcc** can be used to implement co-operative threads