Today we will learn about...

- Continuations
- Continuation-Passing Style (CPS)
- Encoding exceptions with CPS
- Handling exceptions in Racket
- Yield

Other references: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Continuations
What is a continuation?

A technique to abstract control flow. It reifies an execution point as a pair that consists of:

- the program state (e.g., the environment)
- the remaining code to run (e.g., the term)

Used to encode

- exceptions
- generators
- coroutines (lightweight threads)
How can we represent continuations?

- continuation-passing style (inversion of control)
- first-class construct (Racket)
Continuation-passing style (CPS)

Q: How do we abstract computation?
Continuation-passing style (CPS)

Q: How do we abstract computation?

A: Inversion of control

<table>
<thead>
<tr>
<th>Hollywood principle: Don’t call us, we’ll call you.</th>
</tr>
</thead>
<tbody>
<tr>
<td>• the objective is to have control over where a function returns to (its continuation)</td>
</tr>
<tr>
<td>• make <em>returning a value</em> a function call</td>
</tr>
</tbody>
</table>

Direct style

```
(define (f x) (+ x 2))
```

CPS

```
(define (f x ret) (ret (+ x 2)))
```
Where have we seen CPS?

Remember when we implemented the tail-recursive optimization?

Before

```
(define (map f l)
  (cond [(empty? l) l]
        [(else (cons (f (first l)) (map f (rest l))))]))
```

After

```
(define (map f l)
  (define (map-iter l accum)
    (cond [(empty? l) (accum l)]
          [(else (map-iter (rest l) (lambda (x) (accum (cons (f (first l)) x))))))]
    (map-iter l (lambda (x) x))))
```

Function map-iter is the CPS-version of map!
Encoding exceptions with CPS

\[
\text{(define (safe-/) x y)} \\
\hspace{1cm} \text{(lambda (ok err)} \\
\hspace{2cm} \text{(cond [(= 0 y) (err 'division-by-zero)])} \\
\hspace{3cm} \text{[else (ok (/ x y)))]})\]
\]

Example 1

; Print to standard-output if OK and throw an exception if not
((safe-/ 2 1) display error)
; error: division-by-zero
((safe-/ 2 0) display error)

Example 2

- How can we chain two divisions together?

(/ (/ 10 2) 3)
Monadic Continuation-Passing Style
Exceptions Monadic+CPS

; Returns x via the return function
(define (return x)
  (lambda (ret err) (ret x)))

; Returns x via the error function
(define (raise x)
  (lambda (ret err) (err x)))

; Monadic-bind on CPS-style code
(define (cps-bind o1 o2)
  (lambda (ret err) (o1 (lambda (res) ((o2 res) ret err)) err)))

; The try-catch operation
(define (try o1 o2)
  (lambda (ret err) (o1 ret (lambda (res) ((o2 res) ret err))))))

Bind

bind runs o1 and the ok-continuation of o1 is running o2

Try

try runs o1 and the error-continuation is running o2
Revisiting safe-division with monadic API

Thanks to functional programming and monads, we can easily design try-catch on top of a regular computation.

```
(define (&/ x y)
  (cond [(= 0 y) (raise 'division-by-zero)]
        [else (return (/ x y))]))
```
Examples

; 1. Run a division by zero and get an exception
(run? (/ 1 0) (cons 'error 'division-by-zero))

; 2. Run a division by zero and use try-catch to return OK
(run?
  (try
    (/ 1 0)
    (lambda (err) (return 10)))
  (cons 'ok 10))

; 3. Use bind in a more intricate computation
(run?
  (do
    x ← (/ 3 4)
    (try
      (/ x 0)
      (lambda (err) (return 10))))
  (cons 'ok 10))
Exceptions in Racket
How do we catch exception in Racket?

We must use the `with-handler` construct that takes the exception type, and the code that is run when the exception is raised.

```racket
#lang racket
(define (on-err e)
  ; Instead of returning what we were doing, just return #f
  #f
)(with-handlers ([exn:fail:contract:divide-by-zero? on-err]) (/ 1 0))
```
First-class continuations in Racket
First-class support continuations in Racket

Inversion of control

\((\text{call/cc} \ f)\) captures the surrounding code as a \textbf{continuation}, and passes that continuation to function \(f\).

\[
(+\ 1\ 2\ (\text{call/cc}\ f)\ 4\ 5)
\]

becomes

\[
(f\ (\lambda(x)\ (+\ 1\ 2\ x\ 4\ 5)))
\]

Recommended reading

- Many examples using call/cc
Yield

Another way to write streams

(Or, returning streams of values)
Yield: abstracting lazy evaluation

yield allows generalizing returning a finite stream of values (rather than just one). yield actually returns a value, so the caller can interact with the caller. In the following example, yield allows processing multiple files ensuring the garbage collector does not load everything to memory eagerly.

```python

def parse_file(filename):
    # ...
    for root in xml:
        tree = ExecTree(ExecNode(root, resolver=resolver))  # load a possibly big file
        yield tree
    del tree  # garbage collect the memory

## User code
for xml in parse_file(somefile):
    handle(xml)  # handle the xml object
```
Implementing yield

Let us implement yield in Racket!

- **Yield: Mainstream Delimited Continuations.** TPDC. 2011

Papers are still being published in top Programming Language conferences on this subject:

- **Theory and Practice of Coroutines with Snapshots.** ECOOP. 2018
Yield summary

1. Run a CPS computation normally until `(yield x)`
2. The execution of `(yield x)` should suspend the current execution
3. There must exist an execution context that can run suspenable computations
Implementation

Yield is a regular CPS-monadic operation but it returns a suspended object, rather than using `ok` or `err`.

```scheme
(struct susp (value ok) #:transparent)
(define (yield v)
  (lambda (ok err) (susp v ok)))
(define (resume s)
  ((susp-ok s) (void)))
```

(Demo...)