Continuation-Passing Style
CPS

Moving away from Scheme as the Host Language

Regular Factorial, Hosted in Scheme

```
(define fact
    (lambda (n)
        (if (= n 0)
            1
            (* n (fact (- n 1))))))
```

Regular Factorial, Hosted in C#

```
public class Fact {

    public static int fact(int n) {
        if (n == 0)
            return 1;
        else
            return (n * fact(n – 1));
    }

    public static void Main() {
        System.Console.WriteLine(fact(100));
    }
}
```

Recursion and the Stack

In languages (other than Scheme),
recursion will overflow the "stack"

Solutions:

1) Don't use recursion, OR
2) Work around recursion in the host language

Abstraction to the Rescue!

Let's abstract the problem, so we can avoid using the stack.

Recursion is a way of keeping track of
what to do next.

Regular Factorial, Hosted in Scheme

```
(define fact
    (lambda (n)
        (if (= n 0)
            1
            (* n (fact (- n 1))))))
```

Regular Factorial, Hosted in Scheme

```
(define fact
    (lambda (n)
        (if (= n 0)
            1
            (* n (fact (- n 1))))))
```

Continuations

A data structure that represents
what is left to do.

Continuations

We will represent continuations via functions.

These continuations will take one parameter, v, which is the result.

All recursive functions will take a continuation, k, and will apply it when they have a result.

Regular Factorial, Hosted in Scheme

```
(define fact
    (lambda (n)
        (if (= n 0)
            1
            (* n (fact (- n 1))))))
```

CPS Factorial, Hosted in Scheme

```
(define fact-cps
    (lambda (n k)
        (if (= n 0)
            (k 1)
            (fact-cps (- n 1)
                (lambda (v)
                    (k (* n v)))))))
```

CPS Factorial, Hosted in Scheme

```
(define fact-cps
    (lambda (n k)
        (if (= n 0)
            (k 1)
            (fact-cps (- n 1)
                (lambda (v)
                    (k (* n v)))))))


(fact-cps 5 (lambda (v) v))
```

You try!

```
(define length
    (lambda (lyst)
        (cond
            ((null? lyst) 0)
            (else (+ 1 (length (cdr lyst)))))))


(length '(1 2 3 4 5))
```

Length in CPS

```
(define length-cps
    (lambda (lyst k)
        (cond
            ((null? lyst) (k 0))
            (else (length-cps (cdr lyst)
                     (lambda (v)
                        (k (+ 1 v))))))))
```

```
(length-cps '(1 2 3 4 5) (lambda (v) v))
```

CPS, Hosted in other languages

Not all languages have closures,
so we'll develop a **data structure**
representation of closures

Length in CPS

```
(define length-cps
    (lambda (lyst k)
        (cond
            ((null? lyst) (k 0))
            (else (length-cps (cdr lyst)
                           (lambda (v)
                              (k (+ 1 v)))))))))


(length-cps '(1 2 3 4 5) (lambda (v) v))
```

Length in CPS, with DS

```
(define length-cps-ds
    (lambda (lyst k)
        (cond
            ((null? lyst) (apply-cont k 0))
            (else (length-cps-ds (cdr lyst)
                     (make-cont “addem” k))))))
```

```
(length-cps-ds '(1 2 3 4 5) (make-cont “ident”))
```

Length in CPS, with DS

```
(define make-cont list)
(define apply-cont
   (lambda (k v)
      (cond
       ((equal? (car k) “ident”) v)
       ((equal? (car k) “addem”)
        (apply-cont (cadr k) (+ v 1))))))
(define length-cps-ds
    (lambda (lyst k)
        (cond
            ((null? lyst) (apply-cont k 0))
            (else (length-cps-ds (cdr lyst)
                          (make-cont “addem” k))))))
(length-cps-ds '(1 2 3 4 5) (make-cont “ident”))
```

Length in CPS, with DS

```
(define make-cont list)
(define apply-cont
   (lambda (k v)
      (cond
       ((equal? (car k) “ident”) v)
       ((equal? (car k) “addem”)
        (apply-cont (cadr k) (+ v 1))))))
(define length-cps-ds
    (lambda (lyst k)
       (cond
           ((null? lyst) (apply-cont k 0))
           (else (length-cps-ds (cdr lyst)
                         (make-cont “addem” k))))))
(length-cps-ds '(1 2 3 4 5) (make-cont “ident”))
```

Getting rid of Recursion

“A lambda with no parameters,
is just like a GOTO”

Getting rid of Recursion

```
(define proc
   (lambda ()
      (func1)
      (func2)
      (func3)
      ...
      (proc)))
```

Getting rid of Recursion

```
(define proc
   (lambda ()
      (func1)
      (func2)
      (func3)
      ...
      (proc)))
```

```
proc:
      func1();
      func2();
      func3();
      ...
      GOTO proc
```

Getting rid of Recursion

```
(define proc
   (lambda (a b c)
      (func1 a)
      (func2 b)
      (func3 c)
      ...
      (proc a b c)))
```

Getting rid of Recursion

```
(define reg-a 0)
(define reg-b 1)
(define reg-c 2)

(define proc
   (lambda ()
      (func1 reg-a)
      (func2 reg-b)
      (func3 reg-c)
      ...
      (set! reg-a (+ reg-a 1)
      (set! reg-b (- reg-a 1)
      (set! reg-c (* reg-a 6)
      (proc)))
```

Register Machine (RM)

```
(define reg-a 0)
(define reg-b 1)
(define reg-c 2)

(define proc
   (lambda ()
      (func1 reg-a)
      (func2 reg-b)
      (func3 reg-c)
      ...
      (set! reg-a (+ reg-a 1))
      (set! reg-b (- reg-a 1))
      (set! reg-c (* reg-a 6))
      (proc)))
```

```
reg-a = 0
reg-b = 1
reg-c = 2

proc:

   func1();
   func2();
   func3();
   ...
   reg-a = reg-a + 1;
   reg-b = reg-a - 1;
   reg-c = reg-a * 6;
   GOTO proc
```