

Continuations

6.S184 - Zombies drink caffeinated 6.001

Alex Vandiver

Massachusetts Institute of Technology

Lecture 7

Deferred operations

```
(define the-cons (cons 1 #f))  
(set-cdr! the-cons the-cons)  
  
(define (run-in-circles l)  
  (+ (run-in-circles (cdr l))))
```

Deferred operations

```
(define the-cons (cons 1 #f))  
(set-cdr! the-cons the-cons)  
  
(define (run-in-circles l)  
  (+ (run-in-circles (cdr l))))
```

.

Deferred operations

```
(define the-cons (cons 1 #f))  
(set-cdr! the-cons the-cons)  
  
(define (run-in-circles l)  
  (+ (run-in-circles (cdr l))))
```

..

Deferred operations

```
(define the-cons (cons 1 #f))  
(set-cdr! the-cons the-cons)  
  
(define (run-in-circles l)  
  (+ (run-in-circles (cdr l))))
```

..“The program ran out of memory”

Tail recursion in action

```
(define the-cons (cons 1 #f))  
(set-cdr! the-cons the-cons)  
  
(define (run-in-circles l)  
  (run-in-circles (cdr l)))
```

Tail recursion in action

```
(define the-cons (cons 1 #f))  
(set-cdr! the-cons the-cons)
```

```
(define (run-in-circles l)  
  (run-in-circles (cdr l)))
```

.

Tail recursion in action

```
(define the-cons (cons 1 #f))  
(set-cdr! the-cons the-cons)
```

```
(define (run-in-circles l)  
  (run-in-circles (cdr l)))
```

..

Tail recursion in action

```
(define the-cons (cons 1 #f))  
(set-cdr! the-cons the-cons)
```

```
(define (run-in-circles l)  
  (run-in-circles (cdr l)))
```

...

Tail recursion in action

```
(define the-cons (cons 1 #f))  
(set-cdr! the-cons the-cons)
```

```
(define (run-in-circles l)  
  (run-in-circles (cdr l)))
```

.....

Tail recursion in action

```
(define the-cons (cons 1 #f))  
(set-cdr! the-cons the-cons)
```

```
(define (run-in-circles l)  
  (run-in-circles (cdr l)))
```

.....

Tail recursion in action

```
(define the-cons (cons 1 #f))  
(set-cdr! the-cons the-cons)
```

```
(define (run-in-circles l)  
  (run-in-circles (cdr l)))
```

.....

Tail recursion in action

```
(define the-cons (cons 1 #f))  
(set-cdr! the-cons the-cons)
```

```
(define (run-in-circles l)  
  (run-in-circles (cdr l)))
```

.....

- What if we never had any deferred operations?

Continuations

- What if we never had any deferred operations?
- Instead of *returning a value* with deferred operations, the function is passed a *continuation procedure*, which we call to return a value

Continuations

- What if we never had any deferred operations?
- Instead of *returning a value* with deferred operations, the function is passed a *continuation procedure*, which we call to return a value
- Which means that all function calls are *tail-recursive*

Simple CPS example

```
(define (add-17 x)
  (+ x 17))
```

Simple CPS example

```
(define (add-17 x)
  (+ x 17))
```

```
(define (add-17 x cont)
  (cont (+ x 17)))
```

Factorial in CPS

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

Factorial in CPS

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

```
(define (factorial n cont)
  (if (= n 0)
      (cont 1)
      (factorial (- n 1)
                  (lambda (x) (cont (* n x))))))
```

Factorial in CPS

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

```
(define (factorial n cont)
  (if (= n 0)
      (cont 1)
      (factorial (- n 1)
                  (lambda (x) (cont (* n x))))))
```

```
(factorial 10 (lambda (x) x))
```

Factorial in CPS

```
(define (factorial n cont)
  (if (= n 0)
      (cont 1)
      (factorial (- n 1)
                  (lambda (x) (cont (* n x))))))

(factorial 10 (lambda (x) x))
```

- No deferred operations

Factorial in CPS

```
(define (factorial n cont)
  (if (= n 0)
      (cont 1)
      (factorial (- n 1)
                  (lambda (x) (cont (* n x))))))

(factorial 10 (lambda (x) x))
```

- No deferred operations
- We craft a **new** continuation, based on the previous one, and pass that to our recursive call

Factorial in CPS

```
(define (factorial n cont)
  (if (= n 0)
      (cont 1)
      (factorial (- n 1)
                  (lambda (x) (cont (* n x))))))

(factorial 10 (lambda (x) x))
```

- No deferred operations
- We craft a **new** continuation, based on the previous one, and pass that to our recursive call
- Asks the question, “What will I do with the return value of the recursive call?”

Factorial in CPS

```
(define (factorial n cont)
  (if (= n 0)
      (cont 1)
      (factorial (- n 1)
                  (lambda (x) (cont (* n x))))))

(factorial 10 (lambda (x) x))
```

- No deferred operations
- We craft a **new** continuation, based on the previous one, and pass that to our recursive call
- Asks the question, “What will I do with the return value of the recursive call?”
- “Multiply it by n , and call *my* continuation with that value”

Sum-interval

```
(define (sum-interval a b cont)
  (if (= a b)
      (cont a)
      (sum-interval
       (+ a 1)
       b
       (lambda (x) (cont (+ a x)))))))
```

Append

```
(define (append l1 l2)
  (if (eq? l1 '())
      l2
      (cons (car l1) (append (cdr l1) l2))))
```

Append

```
(define (append l1 l2)
  (if (eq? l1 '())
      l2
      (cons (car l1) (append (cdr l1) l2))))
```

```
(define (cs-append l1 l2 cont)
  (if (eq? l1 '())
      (cont l2)
      (cs-append
       (cdr l1)
       l2
       (lambda (appended-cdr)
         (cons (car l1) appended-cdr)))))
```

Append, done right

```
(define (append l1 l2)
  (if (eq? l1 '())
      l2
      (cons (car l1) (append (cdr l1) l2))))
```

```
(define (cs-append l1 l2 cont)
  (if (eq? l1 '())
      (cont l2)
      (cs-append
       (cdr l1)
       l2
       (lambda (appended-cdr)
         (cont (cons (car l1) appended-cdr)))))))
```

Flatten

```
(define (flatten tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (list tree))
        (else (append (flatten (car tree))
                        (flatten (cdr tree))))))
```

Flatten

```
(define (flatten tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (list tree))
        (else (append (flatten (car tree))
                        (flatten (cdr tree))))))
```

```
(define (cs-flatten tree cont)
  (cond ((null? tree) (cont '()))
        ((not (pair? tree)) (cont (list tree)))
        (else (cs-flatten
                 (car tree)
                 (lambda (car-leaves)
                   (cs-flatten
                    (cdr tree)
                    (lambda (cdr-leaves)
                      (cont
                       (append car-leaves cdr-leaves))))))))))
```

Flatten

```
(define (flatten tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (list tree))
        (else (append (flatten (car tree))
                       (flatten (cdr tree))))))
```

```
(define (cs-flatten tree cont)
  (cond ((null? tree) (cont '()))
        ((not (pair? tree)) (cont (list tree)))
        (else (cs-flatten
                (car tree)
                (lambda (car-leaves)
                  (cs-flatten
                   (cdr tree)
                   (lambda (cdr-leaves)
                     (cont
                      (append car-leaves cdr-leaves)
                      ))))))))
```

Flatten

```
(define (flatten tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (list tree))
        (else (append (flatten (car tree))
                        (flatten (cdr tree))))))
```

```
(define (cs-flatten tree cont)
  (cond ((null? tree) (cont '()))
        ((not (pair? tree)) (cont (list tree)))
        (else (cs-flatten
                 (car tree)
                 (lambda (car-leaves)
                   (cs-flatten
                    (cdr tree)
                    (lambda (cdr-leaves)
                      (cont
                       (append car-leaves cdr-leaves)
                       ))))))))
```

Flatten

```
(define (flatten tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (list tree))
        (else (append (flatten (car tree))
                        (flatten (cdr tree))))))
```

```
(define (cs-flatten tree cont)
  (cond ((null? tree) (cont '()))
        ((not (pair? tree)) (cont (list tree)))
        (else (cs-flatten
                 (car tree)
                 (lambda (car-leaves)
                   (cs-flatten
                    (cdr tree)
                    (lambda (cdr-leaves)
                      (cont
                       (append car-leaves cdr-leaves))))))))))
```

Flatten

```
(define (flatten tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (list tree))
        (else (append (flatten (car tree))
                        (flatten (cdr tree))))))
```

```
(define (cs-flatten tree cont)
  (cond ((null? tree) (cont '()))
        ((not (pair? tree)) (cont (list tree)))
        (else (cs-flatten
                (car tree)
                (lambda (car-leaves)
                  (cs-flatten
                   (cdr tree)
                   (lambda (cdr-leaves)
                     (cont
                      (append car-leaves cdr-leaves))))))))))
```

Flatten

```
(define (flatten tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (list tree))
        (else (append (flatten (car tree))
                       (flatten (cdr tree))))))
```

```
(define (cs-flatten tree cont)
  (cond ((null? tree) (cont '()))
        ((not (pair? tree)) (cont (list tree)))
        (else (cs-flatten
                (car tree)
                (lambda (car-leaves)
                  (cs-flatten
                   (cdr tree)
                   (lambda (cdr-leaves)
                     (cont
                      (append car-leaves cdr-leaves)
                      ))))))))
```

- Continuation-passing style is also very useful in controlling program flow

- Continuation-passing style is also very useful in controlling program flow
- Error handling and exceptions is a classic case:

- Continuation-passing style is also very useful in controlling program flow
- Error handling and exceptions is a classic case:

```
(define (divide a b success fail)
  (if (= b 0)
      (fail "divide-by-zero")
      (success (/ a b))))
```

Continuations in the interpreter

We can write a Scheme interpreter in continuation-passing style

Continuations in the interpreter

We can write a Scheme interpreter in continuation-passing style

```
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (if (eq? input '**quit**)
        'c-eval-done
        (c-eval
         input
         the-global-environment
         (lambda (output)
           (announce-output output-prompt)
           (display output)
           (driver-loop)))))))
```

```
(define (c-eval exp env cont)
  (cond ((self-evaluating? exp)
        (cont exp))
        ((variable? exp)
         (cont (lookup-variable-value exp env)))
        ((quoted? exp)
         (cont (text-of-quotation exp)))
        ((assignment? exp)
         (eval-assignment exp env cont))
        ((definition? exp)
         (eval-definition exp env cont))
        ((if? exp) (eval-if exp env cont))
        ((lambda? exp)
         (cont (make-procedure (lambda-parameters exp)
                                (lambda-body exp) env)))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env cont))
        ((cond? exp)
         (c-eval (cond->if exp) env cont))
        ...
```

```
(define (c-eval exp env cont)
  (cond ((self-evaluating? exp)
        (cont exp))
        ((variable? exp)
         (cont (lookup-variable-value exp env)))
        ((quoted? exp)
         (cont (text-of-quotation exp)))
        ((assignment? exp)
         (eval-assignment exp env cont))
        ((definition? exp)
         (eval-definition exp env cont))
        ((if? exp) (eval-if exp env cont))
        ((lambda? exp)
         (cont (make-procedure (lambda-parameters exp)
                                (lambda-body exp) env)))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env cont))
        ((cond? exp)
         (c-eval (cond->if exp) env cont))
        ...
```

```
(define (c-eval exp env cont)
  (cond ((self-evaluating? exp)
        (cont exp))
        ((variable? exp)
         (cont (lookup-variable-value exp env)))
        ((quoted? exp)
         (cont (text-of-quotation exp)))
        ((assignment? exp)
         (eval-assignment exp env cont))
        ((definition? exp)
         (eval-definition exp env cont))
        ((if? exp) (eval-if exp env cont))
        ((lambda? exp)
         (cont (make-procedure (lambda-parameters exp)
                                (lambda-body exp) env)))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env cont))
        ((cond? exp)
         (c-eval (cond->if exp) env cont))
        ...
```

```
(define (c-eval exp env cont)
  (cond ((self-evaluating? exp)
        (cont exp))
        ((variable? exp)
         (cont (lookup-variable-value exp env)))
        ((quoted? exp)
         (cont (text-of-quotation exp)))
        ((assignment? exp)
         (eval-assignment exp env cont))
        ((definition? exp)
         (eval-definition exp env cont))
        ((if? exp) (eval-if exp env cont))
        ((lambda? exp)
         (cont (make-procedure (lambda-parameters exp)
                                (lambda-body exp) env)))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env cont))
        ((cond? exp)
         (c-eval (cond->if exp) env cont))
        ...
```

```
(define (eval-if exp env cont)
  (c-eval
   (if-predicate exp) env
   (lambda (test-value)
     (if test-value
         (c-eval (if-consequent exp) env cont)
         (c-eval (if-alternative exp) env cont))))))
```

```
(define (eval-if exp env cont)
  (c-eval
   (if-predicate exp) env
   (lambda (test-value)
     (if test-value
         (c-eval (if-consequent exp) env cont)
         (c-eval (if-alternative exp) env cont))))))
```

```
(define (eval-sequence exps env cont)
  (if (last-exp? exps)
      (c-eval (first-exp exps) env cont)
      (c-eval (first-exp exps) env
               (lambda (ignored)
                 (eval-sequence
                  (rest-exps exps)
                  env cont))))))
```

Continuations with the interpreter

- What if the evaluator made its continuations available to the language?

Continuations with the interpreter

- What if the evaluator made its continuations available to the language?
- `call-with-current-continuation` procedure

Continuations with the interpreter

- What if the evaluator made its continuations available to the language?
- `call-with-current-continuation` procedure

```
;; Special form for evaluator
```

```
(define (eval-call-with-current-continuation exp env cont)
  (c-eval
   (call/cc-proc exp) env
   (lambda (proc-to-call)
     (c-apply proc-to-call
              (list (make-continuation cont))
              cont))))
```

```
;; in c-apply
```

```
((continuation? procedure)
 (apply (continuation-internal-cont procedure)
        arguments))
```

call/cc example

```
(+ (* 3 (call-with-current-continuation
        (lambda (cont)
          (cont 5))))
  10)
```

call/cc example

```
(+ (* 3 (call-with-current-continuation
        (lambda (cont)
          (cont 5))))
  10)
; => 25
```

call/cc example

```
(+ (* 3 (call-with-current-continuation
        (lambda (cont)
          (cont 5))))))
```

10)

; => 25

```
(define c #f)
```

```
(+ (* 3 (call-with-current-continuation
        (lambda (cont)
          (set! c cont)
          (cont 5))))))
```

10)

call/cc example

```
(+ (* 3 (call-with-current-continuation
        (lambda (cont)
          (cont 5))))
  10)
```

```
; => 25
```

```
(define c #f)
```

```
(+ (* 3 (call-with-current-continuation
        (lambda (cont)
          (set! c cont)
          (cont 5))))
  10)
```

```
; => 25
```

call/cc example

```
(+ (* 3 (call-with-current-continuation
        (lambda (cont)
          (cont 5))))
  10)
; => 25

(define c #f)
(+ (* 3 (call-with-current-continuation
        (lambda (cont)
          (set! c cont)
          (cont 5))))
  10)
; => 25
(c 6)
```

call/cc example

```
(+ (* 3 (call-with-current-continuation
        (lambda (cont)
          (cont 5))))
  10)
; => 25

(define c #f)
(+ (* 3 (call-with-current-continuation
        (lambda (cont)
          (set! c cont)
          (cont 5))))
  10)
; => 25
(c 6)
; => 28
```

call/cc example

```
(+ (* 3 (call-with-current-continuation
        (lambda (cont)
          (cont 5))))
  10)
; => 25

(define c #f)
(+ (* 3 (call-with-current-continuation
        (lambda (cont)
          (set! c cont)
          (cont 5))))
  10)
; => 25
(c 6)
; => 28
(+ 100 (c 6))
```

call/cc example

```
(+ (* 3 (call-with-current-continuation
        (lambda (cont)
          (cont 5))))
  10)
; => 25

(define c #f)
(+ (* 3 (call-with-current-continuation
        (lambda (cont)
          (set! c cont)
          (cont 5))))
  10)
; => 25
(c 6)
; => 28
(+ 100 (c 6))
; => 28
```

call/cc explained

- `call-with-current-continuation` (or `call/cc`, as it is usefully shortened to) takes a procedure as an argument, and passes it the evaluator's current continuation

call/cc explained

- `call-with-current-continuation` (or `call/cc`, as it is usefully shortened to) takes a procedure as an argument, and passes it the evaluator's current continuation
- The return value of `call/cc` is the same as the return value of the procedure

call/cc explained

- `call-with-current-continuation` (or `call/cc`, as it is usefully shortened to) takes a procedure as an argument, and passes it the evaluator's current continuation
- The return value of `call/cc` is the same as the return value of the procedure
- ... or the procedure could just call the continuation it was given. *Which is exactly identical in meaning!*

call/cc explained

- `call-with-current-continuation` (or `call/cc`, as it is usefully shortened to) takes a procedure as an argument, and passes it the evaluator's current continuation
- The return value of `call/cc` is the same as the return value of the procedure
- ... or the procedure could just call the continuation it was given. *Which is exactly identical in meaning!*
- The continuation of the `call/cc` expression, the continuation of the procedure that it calls, and the **value** that it passes as an argument to that procedure, are all the same!

Storing continuations

- Stored continuations can be saved away to “jump back” at any later point in time

Storing continuations

- Stored continuations can be saved away to “jump back” at any later point in time

```
(define cont #f)
(if (call/cc (lambda (c)
              (set! cont c)
              #t)))
  'something
  'other-thing)
```

Storing continuations

- Stored continuations can be saved away to “jump back” at any later point in time

```
(define cont #f)
(if (call/cc (lambda (c)
              (set! cont c)
              #t))
    'something
    'other-thing)
; => 'something
```

Storing continuations

- Stored continuations can be saved away to “jump back” at any later point in time

```
(define cont #f)
(if (call/cc (lambda (c)
              (set! cont c)
              #t))
    'something
    'other-thing)
; => 'something
(cont #f)
```

Storing continuations

- Stored continuations can be saved away to “jump back” at any later point in time

```
(define cont #f)
(if (call/cc (lambda (c)
              (set! cont c)
              #t))
    'something
    'other-thing)
; => 'something
(cont #f)
; => 'other-thing
```

```
(define (fib-func)
  (let ((prev 0)
        (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      prev)
    loop))
```

```
(define (fib-func)
  (let ((prev 0)
        (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      prev)
    loop))
(define test (fib-func))
```

```
(define (fib-func)
  (let ((prev 0)
        (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      prev)
    loop))
(define test (fib-func))
(test) ; => 1
```

```
(define (fib-func)
  (let ((prev 0)
        (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      prev)
    loop))
(define test (fib-func))
(test) ; => 1
(test) ; => 1
```

```
(define (fib-func)
  (let ((prev 0)
        (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      prev)
    loop))
(define test (fib-func))
(test) ; => 1
(test) ; => 1
(test) ; => 2
```

```
(define (fib-func)
  (let ((prev 0)
        (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      prev)
    loop))
(define test (fib-func))
(test) ; => 1
(test) ; => 1
(test) ; => 2
(test) ; => 3
```

```
(define (fib-func)
  (let ((prev 0)
        (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      prev)
    loop))
(define test (fib-func))
(test) ; => 1
(test) ; => 1
(test) ; => 2
(test) ; => 3
(test) ; => 5
```

```
(define resume #f)
(define (fib-cont)
  (let ((prev 0)
        (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      (if (call/cc
          (lambda (c)
            (set! resume (lambda () (c #f)))
            (c #t)))
          prev
          (loop)))
    (loop)))
```

```
(define resume #f)
(define (fib-cont)
  (let ((prev 0)
        (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      (if (call/cc
          (lambda (c)
            (set! resume (lambda () (c #f)))
            (c #t)))
          prev
          (loop)))
    (loop)))
(fib-cont) ; => 1
```

```

(define resume #f)
(define (fib-cont)
  (let ((prev 0)
        (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      (if (call/cc
          (lambda (c)
            (set! resume (lambda () (c #f)))
            (c #t)))
          prev
          (loop))))
    (loop)))
(fib-cont) ; => 1
(resume)   ; => 1

```

```

(define resume #f)
(define (fib-cont)
  (let ((prev 0)
        (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      (if (call/cc
           (lambda (c)
             (set! resume (lambda () (c #f)))
             (c #t)))
          prev
          (loop))))
    (loop)))
(fib-cont) ; => 1
(resume)   ; => 1
(resume)   ; => 2

```

```

(define resume #f)
(define (fib-cont)
  (let ((prev 0)
        (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      (if (call/cc
          (lambda (c)
            (set! resume (lambda () (c #f)))
            (c #t)))
          prev
          (loop))))
    (loop)))
(fib-cont) ; => 1
(resume)  ; => 1
(resume)  ; => 2
(resume)  ; => 3

```

```

(define resume #f)
(define (fib-cont)
  (let ((prev 0)
        (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      (if (call/cc
          (lambda (c)
            (set! resume (lambda () (c #f)))
            (c #t)))
          prev
          (loop))))
    (loop)))
(fib-cont) ; => 1
(resume) ; => 1
(resume) ; => 2
(resume) ; => 3
(resume) ; => 5

```

- Save the continuation, return **true** now

- Save the continuation, return **true** now
- But call the continuation with **false** again, sometime in the future, to take the other branch

- Save the continuation, return **true** now
- But call the continuation with **false** again, sometime in the future, to take the other branch
- In this case, resumes the loop!

- Save the continuation, return **true** now
- But call the continuation with **false** again, sometime in the future, to take the other branch
- In this case, resumes the loop!
- This pattern is known as a **coroutine**

- Save the continuation, return **true** now
- But call the continuation with **false** again, sometime in the future, to take the other branch
- In this case, resumes the loop!
- This pattern is known as a **coroutine**
- Poor man's threading (running multiple things at once)

- Save the continuation, return **true** now
- But call the continuation with **false** again, sometime in the future, to take the other branch
- In this case, resumes the loop!
- This pattern is known as a **coroutine**
- Poor man's threading (running multiple things at once)
- ... but we can do better...

Co-operative multithreading

- Only one bit of code can run at once, but we have multiple tasks to do

Co-operative multithreading

- Only one bit of code can run at once, but we have multiple tasks to do
- Make each task declare when it's done doing some computation, and then swap

Co-operative multithreading

- Only one bit of code can run at once, but we have multiple tasks to do
- Make each task declare when it's done doing some computation, and then swap
- “Co-operative” because tasks need to declare when they want to let someone else have a turn

Co-operative multithreading

- Only one bit of code can run at once, but we have multiple tasks to do
- Make each task declare when it's done doing some computation, and then swap
- “Co-operative” because tasks need to declare when they want to let someone else have a turn
- Used by Mac OS 9, Windows 3.1

Co-operative multithreading plan

- Keep a list of `threads` (thunks which call continuations)

Co-operative multithreading plan

- Keep a list of `threads` (thunks which call continuations)
- The first thread in the list is always the currently-running one.

Co-operative multithreading plan

- Keep a list of `threads` (thunks which call continuations)
- The first thread in the list is always the currently-running one.
- When we `yield`, we store the current continuation in the list, rotate the list of continuations by one, and jump into the new first continuation.

Co-operative multithreading plan

- Keep a list of `threads` (thunks which call continuations)
- The first thread in the list is always the currently-running one.
- When we `yield`, we store the current continuation in the list, rotate the list of continuations by one, and jump into the new first continuation.
- When we `fork`, add a new thunk to end of the list, which calls the fork'd code then removes itself from the list of threads.

Basic thread list abstraction

```
(define threads '())  
(define (reset-threads!)  
  (set! threads (list 'threads)))  
(define (threads-list) (rest threads))  
(define (current-thread) (first (threads-list)))
```

Adding, saving, and removing threads

```
(define (add-thread! thunk)
  (set-cdr! (last-pair threads)
            (list (lambda ()
                    (thunk)
                    (drop-and-schedule!))))))
```

```
(define (save-thread! thunk)
  (set-car! (threads-list)
            thunk))
```

```
(define (pop-thread!)
  (set-cdr! threads (rest (threads-list))))
```

Scheduling threads

```
(define (rotate-threads! keep)
  (if keep (set-cdr! (last-pair threads)
                    (list (current-thread))))
  (pop-thread!)
  (if (null? (threads-list))
      threads-done
      (current-thread)))

(define (drop-and-schedule!)
  ((rotate-threads! #f)))

(define (schedule)
  ((rotate-threads! #t)))
```

Setup

```
(define threads-done #f)
(define (start-threads thunk)
  (reset-threads!)
  (add-thread! thunk)
  (if (call/cc
      (lambda (cont)
        (set! threads-done (lambda () (cont #t)))
        #f))
      'all-done
      (schedule))))
```

Fork and yield

```
(define fork add-thread!)  
  
(define (yield)  
  (let ((cont #f))  
    (if (call/cc (lambda (k) (set! cont k) #f))  
        'yield-done  
        (begin  
          (save-thread! (lambda () (cont #t)))  
          (schedule))))))
```

```
(define (test)
  (define (h1 n thread-name)
    (if (= n 0) 0
        (begin
          (display thread-name)
          (display " - ")
          (display n)
          (newline)
          (yield)
          (h1 (- n 1) thread-name))))
  (fork (lambda () (h1 5 'thread1)))
  (fork (lambda () (h1 7 'thread2)))
  (fork (lambda () (h1 9 'thread3))))

(start-threads test)
```

Results

```
> (start-threads test)
thread1 - 5
thread2 - 7
thread3 - 9
thread1 - 4
thread2 - 6
thread3 - 8
thread1 - 3
thread2 - 5
thread3 - 7
thread1 - 2
thread2 - 4
thread3 - 6
thread1 - 1
thread2 - 3
thread3 - 5
thread2 - 2
thread3 - 4
thread2 - 1
thread3 - 3
thread3 - 2
thread3 - 1
all-done
>
```

- Inter-process communication is often done through events

Asynchronous computing

- Inter-process communication is often done through events
- Events are a good model for user interaction

Asynchronous computing

- Inter-process communication is often done through events
- Events are a good model for user interaction
- Which leads to graphical interfaces

Asynchronous computing

- Inter-process communication is often done through events
- Events are a good model for user interaction
- Which leads to graphical interfaces
- Which DrScheme has an implementation of

```
(require mred) ; DrScheme graphical toolkit
(define frame
  (instantiate frame% ("Hello world")))
(define msg
  (instantiate message%
    ("No events so far..." frame)))

(instantiate
  button% ()
  (label "Click Me")
  (parent frame)
  (callback (lambda (button event)
              (send msg set-label "Button click"))))

(send frame show #t)
```

A practical example

- User interfaces service requests

A practical example

- User interfaces service requests
- Often waiting on responses from the user or other processes

A practical example

- User interfaces service requests
- Often waiting on responses from the user or other processes
- Would prefer to run as a single thread, so CPS and event processing would be excellent

A practical example

- User interfaces service requests
- Often waiting on responses from the user or other processes
- Would prefer to run as a single thread, so CPS and event processing would be excellent
- Would love to have a language with dynamic scoping, procedures as first-class objects

A practical example

- User interfaces service requests
- Often waiting on responses from the user or other processes
- Would prefer to run as a single thread, so CPS and event processing would be excellent
- Would love to have a language with dynamic scoping, procedures as first-class objects
- If only we had a good language for this, which also came with a UI toolkit. . .

A practical example

- User interfaces service requests
- Often waiting on responses from the user or other processes
- Would prefer to run as a single thread, so CPS and event processing would be excellent
- Would love to have a language with dynamic scoping, procedures as first-class objects
- If only we had a good language for this, which also came with a UI toolkit. . .
- We do:

A practical example

- User interfaces service requests
- Often waiting on responses from the user or other processes
- Would prefer to run as a single thread, so CPS and event processing would be excellent
- Would love to have a language with dynamic scoping, procedures as first-class objects
- If only we had a good language for this, which also came with a UI toolkit. . .
- We do: **Javascript**