Deferred operations

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

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

(run-in-circles the-cons)

."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)))

(run-in-circles the-cons)

- 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))

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

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))

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)
  (if (= a b)
    a
    (+ a (sum-interval (+ a 1) b))))

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

Append, done right

(define (append L1 L2)
  (if (null? L1)
    L2
    (cons (car L1) (append (cdr L1) L2))))

(define (cs-append L1 L2 cont)
  (if (null? L1)
    (cont L2)
    (cs-append
      (cdr L1)
      L2
      (lambda (appended-cdr)
        (cont (cons (car L1) appended-cdr))))))
### Flatten

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

### Control flow

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

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

- Also asynchronous procedure calls

### Continuations in the interpreter

We can write a Scheme interpreter in continuation-passing style:

```scheme
(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))))))
```

```scheme
(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))
        ...
```

---

**Note:** The above text is a translation of the provided image content. It includes code snippets and explanations related to Scheme programming, continuation-passing style, and interpreter design. The text is structured to maintain the logical flow and readability of the original content.
(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)))))))

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

- **call-with-current-continuation** (a.k.a. call/cc)

  ;; 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)
  ; => 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
- 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

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

Mike Phillips <mpp> (MIT)
Continuations
Lecture 7A 17 / 21

```scheme
(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) ; => 2
(test) ; => 3
(test) ; => 5
```

Mike Phillips <mpp> (MIT)
Continuations
Lecture 7A 18 / 21

```scheme
(define resume 'uninitialized)
(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)))
(fib-cont) ; => 1
(resume) ; => 1
(resume) ; => 2
(resume) ; => 3
(resume) ; => 5
```

Mike Phillips <mpp> (MIT)
Continuations
Lecture 7A 19 / 21

Coroutines

- 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
- 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