# Advanced Scheme Techniques

## Some Naughty Bits

Jeremy Brown

January 12 & 14, 2004

# Acknowledgements

Jonathan Bachrach, Alan Bawden, Chris Hanson, Neel Krishnaswami, and Greg Sullivan offered many helpful suggestions on an earlier version of this course.

These slides draw on works by

Hal Abelson, Alan Bawden, Chris Hanson, Paul Graham, Oleg Kiselyov, Neel Krishnaswami, Al Petrofsky, Jonathan Rees, Dorai Sitaram, Gerry Sussman, Julie Sussman, and the R5RS authors group

Thanks also to Scheme Boston, the Boston-area Scheme User's Group.

And of course to SIPB, for organizing.

All errors are, of course, my fault alone.

# Advanced Scheme

# Day 2:

# Continuations

## Scheme Requests for Implementation (SRFIs)

Several of the examples today will refer to SRFIs.

The SRFI documents represent the Scheme community's de facto, post-R5RS standards

Check them out at http://srfi.schemers.org/

## Anatomy of a Closure

In Scheme, procedures are *closures*.

A closure expects to be invoked with a certain number of arguments.

A closure contains:

- a pointer to some code
- a pointer to an environment

## Closure Example

```
((lambda (n)
   (lambda (x) (+ x n))) 5)  ==> #<procedure object>
```

The procedure object has pointers to::

- the code for adding x and n: `(+ x n)`
- the environment binding n to 5

## Procedure Call

When a function invokes a closure, it a single return value.

```
(define (pairify x y)
  (let ((val (cons x y)))
    val))
```

E.g., pairify expects cons to return a single value.

## Return Information

A function must save information to return a value to its caller:

- a pointer to some code: the return address in the caller's code
- a pointer to an environment: the caller's execution environment

## Return Information

A function must save information to return a value to its caller:

- a pointer to some code: the return address in the caller's code
- a pointer to an environment: the caller's execution environment

This return-information:

# Return Information

A function must save information to return a value to its caller:

- a pointer to some code: the return address in the caller's code
- a pointer to an environment: the caller's execution environment

This return-information:

- looks a lot like a closure (pointers to code and env)...

# Return Information

A function must save information to return a value to its caller:

- a pointer to some code: the return address in the caller's code
- a pointer to an environment: the caller's execution environment

This return-information:

- looks a lot like a closure (pointers to code and env)...
- that expects a single argument (the return value)...

# Return Information

A function must save information to return a value to its caller:

- a pointer to some code: the return address in the caller's code
- a pointer to an environment: the caller's execution environment

This return-information:

- looks a lot like a closure (pointers to code and env)...
- that expects a single argument (the return value)...
- and never returns!

# Continuations

Return-information represents the future path of a program.

Consider an actual closure which:

- expects a single argument, and
- never returns to its caller

Given this closure, we can view returning a value $V$ as calling (k V).

## Continuations

A *continuation* is a closure which:

- represents the “future” of a computation from a given point
- never returns to its caller
- (usually) expects one argument — the value to be returned from the point at which the continuation was created

# A Quick Review of Tail Calls

Consider

```
(lambda (x y) (y x))
```

The lambda will return the value returned by `(y x)` — we call `(y x)` a tail-call.

# A Quick Review of Tail Calls

Consider

```
(lambda (x y) (y x))
```

The lambda will return the value returned by `(y x)` — we call `(y x)` a tail-call.

Since the lambda has done all its work by the time the tail-call is called, its environment, etc., do not need to be preserved.

# A Quick Review of Tail Calls

Consider

```
(lambda (x y) (y x))
```

The lambda will return the value returned by `(y x)` — we call `(y x)` a tail-call.

Since the lambda has done all its work by the time the tail-call is called, its environment, etc., do not need to be preserved.

Scheme implementations are required to support unbounded numbers of active tail calls.

# Normal Factorial

Normal fact:

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

(fact 5)   ==> 120
```

# Normal Factorial

Normal fact:

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

(fact 5)   ==> 120
```

What if we made all the implicit returns into explicit continuation calls? (Continuation-Passing Style)

# Continuation Passing Style (CPS)

```
(define (cps-fact k n)
  (cps-=
   (lambda (eq-n-1)
     (if eq-n-1
         (k 1)
         (cps--
          (lambda (nval)
            (cps-fact
             (lambda (rval)
               (cps-* k n rval)) nval)) n 1)))
   n 1))

(cps-fact (lambda (x) x) 5)  ==> 120
```

Note “inside-out” structure: every call is a tail call!

## CPS call-with-current-continuation

call-with-current-continuation (AKA call/cc) makes the return continuation explicitly available as a closure.

1

## CPS call-with-current-continuation

call-with-current-continuation (AKA call/cc) makes the return continuation explicitly available as a closure.

The CPS version of call/cc is simple:

```
(define (cps-call/cc k func)
  (func k k))
```

1

## CPS call-with-current-continuation

call-with-current-continuation (AKA call/cc) makes the return continuation explicitly available as a closure.

The CPS version of call/cc is simple:

```
(define (cps-call/cc k func)
  (func k k))
```

1

The “normal” version of call/cc is a language primitive.

## CPS call-with-current-continuation

call-with-current-continuation (AKA call/cc) makes the return continuation explicitly available as a closure.

The CPS version of call/cc is simple:

```
(define (cps-call/cc k func)
  (func k k))
```

1

The "normal" version of call/cc is a language primitive.

We need an example...

# Early Return Using call/cc

Contrived example use of call/cc

```
(define evencount 0)


(let ((test 17))
  (call/cc (lambda (return)
             (if (odd? test) (return 5))
             (set! evencount (+ evencount 1))
             7)))
 ==> 5
```

# Early Return Using call/cc

Contrived example use of call/cc

```
(define evencount 0)

(let ((test 17))
  (call/cc (lambda (return)
             (if (odd? test) (return 5))
             (set! evencount (+ evencount 1))
             7)))
 ==> 5
```

`(lambda (return...))` receives a continuation in *return*.

The continuation represents returning a value from the call/cc form.

When the continuation is invoked with the argument 5, the call/cc form immediately returns 5. The set! is never executed!

# Continuations are First Class

Continuations...

- are first-class functions
- can be invoked many times
- can be used to create nearly any control-flow structure

# Multiple-Value Continuations

Scheme limits normal functions to returning a single value.

In CPS-style, it's easy to have multiple-value "return":

```
(define (cps-values k . args)
  (cps-apply k args))
```

...all you need is a continuation (k, above) that accepts multiple values!

# Multiple-Value Continuations

Scheme limits normal functions to returning a single value.

In CPS-style, it's easy to have multiple-value "return":

```
(define (cps-values k . args)
  (cps-apply k args))
```

...all you need is a continuation (k, above) that accepts multiple values!

Scheme provides a language primitive "values" to return multiple values:

```
(lambda (a b)
  (values a b))
```

But how do we get the continuation that can accept them?

# call-with-values

Scheme provides another primitive that works with values. From R5RS:

```
(call-with-values
    (lambda () (values 4 5))   ; producer
  (lambda (a b) (+ a b)))      ; consumer
                               ; (continuation)

 ==> 9
```

call-with-values calls the producer, providing the consumer as its continuation

# call-with-values

Scheme provides another primitive that works with values. From R5RS:

```
(call-with-values
    (lambda () (values 4 5))   ; producer
  (lambda (a b) (+ a b)))      ; consumer
                               ; (continuation)

 ==> 9
```

call-with-values calls the producer, providing the consumer as its continuation

SRFI-11 defines special forms LET-VALUES and LET*-VALUES which hide the call-by-values form

# Control Flow Structures

## Control Flow Structures

We've already seen early-return using continuations. Coming up:

- Exceptions
- Iterators/Co-routining
- Backtracking
- Multi-threading

# Exceptions

# Simple Exception Semantics

Simplest possible scheme:

```
(define (le10-or-bust x)
  (if (> x 10) (throw) x))

(let ((x 17))
  (try (lambda () 5)
       (le10-or-bust x)
       12))
 ==> 5
```

# Simple Exception Semantics

Simplest possible scheme:

```
(define (le10-or-bust x)
  (if (> x 10) (throw) x))

(let ((x 17))
  (try (lambda () 5)
       (le10-or-bust x)
       12))
 ==> 5
```

First argument to “try” is the handler; remainder args are body.

# Simple Exception Semantics

Simplest possible scheme:

```
(define (le10-or-bust x)
  (if (> x 10) (throw) x))

(let ((x 17))
  (try (lambda () 5)
       (le10-or-bust x)
       12))
 ==> 5
```

First argument to “try” is the handler; remainder args are body. If (throw) is not called, the body’s return-value is try’s return-value.

# Simple Exception Semantics

Simplest possible scheme:

```
(define (le10-or-bust x)
  (if (> x 10) (throw) x))

(let ((x 17))
  (try (lambda () 5)
       (le10-or-bust x)
       12))
 ==> 5
```

First argument to "try" is the handler; remainder args are body. If (throw) is not called, the body's return-value is try's return-value. Handler is instantly invoked if (throw) is called while execution is in the try-form. Handler's return-value is then also returned by the try expression.

# Simple Exception Implementation

```
(define top-exception-handler  (lambda () (error "unhandled")))
(define (throw) (top-exception-handler))
```

# Simple Exception Implementation

```
(define top-exception-handler  (lambda () (error "unhandled")))
(define (throw) (top-exception-handler))

(define-syntax try
  (syntax-rules ()
    ((try catch-clause body ...)
     (let* ((result #f)
            (old-handler top-exception-handler)
            (success (call/cc (lambda (cont)
                                (set! top-exception-handler
                                      (lambda () (cont #f)))
                                (set! result (begin body ...))
                                #t))))
       (set! top-exception-handler old-handler)
       (if success result (catch-clause))))))
```

## SRFI-34 Exceptions

SRFI-34 defines a more sophisticated exception-handling suite:

- Thrown exceptions include values
- Exception handlers can dispatch on values
- etc.

Check it out.

# Backtracking

## Backtracking: a Teaser

The “amb” operator always picks an acceptable value:

```
(let ((value (amb 0 1 2 3 4 5 6)))
  (assert (> value 2))
  (assert (even? value))
  value)

==>
```

# Backtracking: a Teaser

The “amb” operator always picks an acceptable value:

```
(let ((value (amb 0 1 2 3 4 5 6)))
  (assert (> value 2))
  (assert (even? value))
  value)

==>  4
```

# Backtracking: a Teaser

The “amb” operator always picks an acceptable value:

```
(let ((value (amb 0 1 2 3 4 5 6)))
  (assert (> value 2))
  (assert (even? value))
  value)

==>  4
```

And you can ask for more:

```
(next)
==> 6
```

# Backtracking: An Application

```
(define (three-dice sumto)
  (let ((die1 (amb 1 2 3 4 5 6))
        (die2 (amb 1 2 3 4 5 6))
        (die3 (amb 1 2 3 4 5 6)))
    (assert (= sumto (+ die1 die2 die3)))
    (list die1 die2 die3)))

(initialize-amb-fail)
(three-dice 4)          ==> (2 1 1)
(next)                  ==> (1 2 1)
(next)                  ==> (1 1 2)
(next)                  ==> ERROR:
                            amb tree exhausted
```

## Amb: Principle of Operation

Amb works by *backtracking*

Think of amb as a glorified exception handler:

1. Pick a value and run forward
2. If no exception is thrown, great
3. If an exception is thrown, pick another value and run forward again

# Amb: Framework

Everything but the definition of amb:

```
(define amb-fail '())

(define (initialize-amb-fail)
  (set! amb-fail
        (lambda (x)
          (error "amb tree exhausted"))))

(define (assert pred)
  (if (not pred) (amb)))

(define (fail) (amb))

(define (next) (amb))
```

Adapted from “Teach yourself Scheme in Fixnum Days (TYSiFD)”, by Dorai Sitaram

# Amb: The Macro

```
(define-syntax amb
  (syntax-rules ()
    ((amb argument ...)
     (let ((old-amb-fail amb-fail))
       (call/cc (lambda (return)
                  (call/cc (lambda (next)
                             (set! amb-fail next)
                             (return argument))) ...
                  (set! amb-fail old-amb-fail)
                  (amb-fail #f)))))))
```

Each ambiguous decision point adds to the stack.

Each failure backtracks to the last decision point.

Adapted from “Teach yourself Scheme in Fixnum Days (TYSiFD)”, by Dorai Sitaram

# bag-of: Getting All the Options

bag-of gives you a list of all acceptable solutions:

```
(bag-of (three-dice 4))
==> ((1 1 2) (1 2 1) (2 1 1))
```

# bag-of: Getting All the Options

bag-of gives you a list of all acceptable solutions:

```
(bag-of (three-dice 4))
==> ((1 1 2) (1 2 1) (2 1 1))
```

And it’s recursive:

```
(bag-of
 (let ((sum (amb 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18)))
   (bag-of (three-dice sum))))

 (let loop ((die 18))
   (if (>= die 3)
       (cons (bag-of (three-dice die)) (loop (- die 1)))
       '())))

==> ((((6 6 6)) ((5 6 6) (6 5 6) (6 6 5)) ((4 6 6) (5 5 6) ....
```

## bag-of: The Macro

```
(define-syntax bag-of
  (syntax-rules ()
    ((bag-of expr)
     (let* ((old-amb-fail amb-fail)
            (result '()))
       (if (call/cc (lambda (ifcondcont)
                      (set! amb-fail ifcondcont)
                      (let ((e expr))
                        (set! result (cons e result))
                        (ifcondcont #t))))
           (amb-fail #f))
       (set! amb-fail old-amb-fail)
       result))))
```

# Iterators

# Traversals

It's easy to traverse a data structure recursively:

```
(define (list-traverse list)
  (if (pair? list)
      (list-traverse (cdr list))))
```

# Traversals

It's easy to traverse a data structure recursively:

```
(define (list-traverse list)
  (if (pair? list)
      (list-traverse (cdr list))))

(define (tree-traverse tree)
  (if (pair? tree)
      (begin
        (tree-traverse (car tree))
        (tree-traverse (cdr tree)))))
```

# Traversals

It's easy to traverse a data structure recursively:

```
(define (list-traverse list)
  (if (pair? list)
      (list-traverse (cdr list))))

(define (tree-traverse tree)
  (if (pair? tree)
      (begin
        (tree-traverse (car tree))
        (tree-traverse (cdr tree)))))
```

Not that these do anything useful

# A List Iterator

```
(define (list-iter list)
  (lambda ()
    (if list
        (let ((value (car list)))
          (set! list (cdr list))
          value)
        '())))
```

# A List Iterator

```
(define (list-iter list)
  (lambda ()
    (if list
        (let ((value (car list)))
          (set! list (cdr list))
          value)
        '())))

(define li (list-iter '(1 2 3)))
(li) ==> 1
(li) ==> 2
(li) ==> 3
(li) ==> ()
```

# A List Iterator

```
(define (list-iter list)
  (lambda ()
    (if list
        (let ((value (car list)))
          (set! list (cdr list))
          value)
        '())))

(define li (list-iter '(1 2 3)))
(li) ==> 1
(li) ==> 2
(li) ==> 3
(li) ==> ()
```

This is pretty clean, but...

# Iterating Over a Tree

```
(define (tree-iter tree)
  (let ((cell-stack (list tree)))
    (lambda ()
      (if cell-stack
          (let loop ((node (pop! cell-stack)))
            (if (pair? node)
                (begin
                  (push! (cdr node) cell-stack)
                  (loop (car node)))
                node))
          '()))))

(define ti (tree-iter '((1 . 2) . (3 . 4))))
(ti) ==> 1  etc.
```

...now we’re keeping a history of the computation in cell-stack!

# Tree Iterator Using Continuations and Macros

We add four lines to the tree-traverse routine:

```
(define (tree-iter tree)
  (with-caller caller loopstate  ; save calling cont.
   (let loop ((node tree))
     (if (pair? node)
         (begin
           (loop (car node))
           (loop (cdr node)))
         (begin                              ; sequence
           (send caller loopstate node)      ; send value
           '())))))                          ; 'done' value
```

Adapted from "Teach yourself Scheme in Fixnum Days (TYSiFD)", by Dorai Sitaram

# Helper Macro: Send

```
(send caller localstate value)
```

Send gives the value to the ’caller’ continuation, storing the current continuation in the localstate variable:

```
(with-caller caller localstate body ...)
```

with-caller saves the calling continuation into caller, constructs the lexical execution environment in which localstate is bound, etc.

# send

```
(define-syntax send
  (syntax-rules ()
    ((send to from value)
     (call/cc
      (lambda (state)
        (set! from (lambda () (state 0)))
        (to value))))))
```

# with-caller

```
(define-syntax with-caller
  (syntax-rules ()
    ((with-caller caller iterator body ...)
     (let ((caller #f))
       (letrec ((iterator
                 (lambda ()
                   body ...)))
         (lambda ()
           (call/cc
            (lambda (caller-cont)
              (set! caller caller-cont)
              (iterator)))))))))
```

# Tree Iterator Expansion I

```
(define (tree-iter-k list)
  (let ((caller #f))     ; caller continuation
    (letrec ((iterator
              (lambda ()
                (let loop ((list list))
                  (if list
                      (begin
                        (call/cc
                         (lambda (iter)
                           (set! iterator (lambda () (iter 0)))
                           (caller (car list))))
                        (loop (cdr list)))
                      (caller '())))))))
    ... more
```

# Tree Iterator Expansion II

```
...
    (lambda ()
      (call/cc
       (lambda (caller-cont)
         (set! caller caller-cont)
         (iterator)))))))
```

# Cooperative Multi-Threading

## Simple Goal

Three routines:

```
(start-scheduling thunk)
```

```
(spawn thunk)
```

```
(yield)
```

- start-scheduling kicks off the threading system running thunk
- spawn may be called to create an additional thread from thunk
- yield may be called by one thread to let others run

# Global State

```
(define thread-set '())

(define scheduler-context #f)
```

## start-scheduling

```
(define (start-scheduling thunk)
  (set! thread-set '())
  (call/cc
   (lambda (scheduler)
     (set! scheduler-context scheduler)
     (spawn thunk)))
  (if (not (empty-stack? thread-set))
      (begin
         ((pop! thread-set))
         (loop)
         (display "**Scheduler exiting**."))))
```

# spawn

```
(define (spawn thunk)
  (push! (lambda () (thunk) (scheduler-context 0))
         thread-set))
```

## yield

```
(define (yield)
  (call/cc
   (lambda (this-thread)
     (if (not (empty-stack? thread-set))
         (let ((next-thread (pop! thread-set)))
           (push! (lambda () (this-thread 0)) thread-set)
           (next-thread))))))
```

# Example Code

```
(start-scheduling
 (lambda ()
   (spawn (lambda ()
            (display "sub-thread")
            (yield)
            (display "more sub-thread")
            (yield)))
   (display "first thread")
   (yield)
   (display "and more first")))
```

# Example Output

```
first thread
sub-thread
and more first
more sub-thread
```

## Homework

Can you figure out how to implement locks in this system?

## Other Continuation-Related Functions

Look these up sometime...

- dynamic-wind
- fluid-let

**The End!**