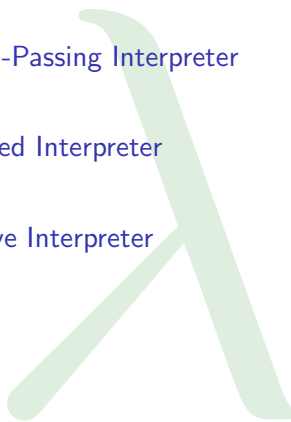


# Part V: Continuation-Passing Interpreters

Marco T. Morazán

Seton Hall University

# Outline

- 
- 1 Continuation-Passing Interpreter
  - 2 A Trampolined Interpreter
  - 3 An Imperative Interpreter
  - 4 Exceptions
  - 5 Threads

# Continuation-Passing Interpreters

Part V:  
Continuation-  
Passing  
Interpreters

Marco T.  
Morazán

Continuation-  
Passing  
Interpreter

A  
Trampoline  
Interpreter

An Imperative  
Interpreter

Exceptions

Threads

- Environments establish the *context* in which each portion of a program is evaluated



# Continuation-Passing Interpreters

Part V:  
Continuation-  
Passing  
Interpreters

Marco T.  
Morazán

Continuation-  
Passing  
Interpreter

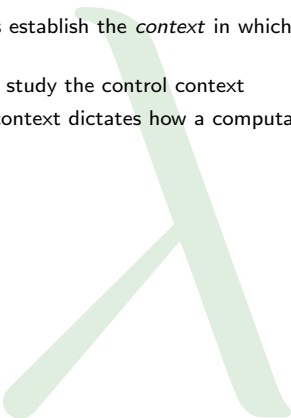
A  
Trampolined  
Interpreter

An Imperative  
Interpreter

Exceptions

Threads

- Environments establish the *context* in which each portion of a program is evaluated
- We shall now study the control context
- The control context dictates how a computation proceeds



# Continuation-Passing Interpreters

Part V:  
Continuation-  
Passing  
Interpreters

Marco T.  
Morazán

Continuation-  
Passing  
Interpreter

A  
Trampolined  
Interpreter

An Imperative  
Interpreter

Exceptions

Threads

- Environments establish the *context* in which each portion of a program is evaluated
- We shall now study the control context
- The control context dictates how a computation proceeds
- Think of a computation as divided in two parts
  - the evaluation of a function call for the value of an argument
  - the rest of the computation that uses the result obtained

# Continuation-Passing Interpreters

Part V:  
Continuation-  
Passing  
Interpreters

Marco T.  
Morazán

Continuation-  
Passing  
Interpreter

A  
Trampolined  
Interpreter

An Imperative  
Interpreter

Exceptions

Threads

- Environments establish the *context* in which each portion of a program is evaluated
- We shall now study the control context
- The control context dictates how a computation proceeds
- Think of a computation as divided in two parts
  - the evaluation of a function call for the value of an argument
  - the rest of the computation that uses the result obtained
- The control context is made explicit (just as environments)

# Continuation-Passing Interpreters

Part V:  
Continuation-  
Passing  
Interpreters

Marco T.  
Morazán

Continuation-  
Passing  
Interpreter

A  
Trampoline  
Interpreter

An Imperative  
Interpreter

Exceptions

Threads

- Environments establish the *context* in which each portion of a program is evaluated
- We shall now study the control context
- The control context dictates how a computation proceeds
- Think of a computation as divided in two parts
  - the evaluation of a function call for the value of an argument
  - the rest of the computation that uses the result obtained
- The control context is made explicit (just as environments)
- A *continuation* is an abstraction for the control context
- A continuation knows how to finish a computation after an intermediate value is computed
- We say that a continuation is applied to an intermediate value to finish the computation

# Continuation-Passing Interpreters

- Consider:  

```
(define (add a b)
  (if (= b 0)
      a
      (+ 1 (add a (- b 1))))))
```



# Continuation-Passing Interpreters

- Consider:

```
(define (add a b)
  (if (= b 0)
      a
      (+ 1 (add a (- b 1))))))
```

- Trace (add 3 3):

```
= (+ 1 (add 3 2))
= (+ 1 (+ 1 (add 3 1)))
= (+ 1 (+ 1 (+ 1 (add 3 0))))
= (+ 1 (+ 1 (+ 1 3)))
= (+ 1 (+ 1 4))
= (+ 1 5)
= 6
```

# Continuation-Passing Interpreters

- Consider:  

```
(define (add a b)
  (if (= b 0)
      a
      (+ 1 (add a (- b 1))))))
```
- Trace (add 3 3):  

```
= (+ 1 (add 3 2))
= (+ 1 (+ 1 (add 3 1)))
= (+ 1 (+ 1 (+ 1 (add 3 0))))
= (+ 1 (+ 1 (+ 1 3)))
= (+ 1 (+ 1 4))
= (+ 1 5)
= 6
```
- Each recursive call comes with the promise that its result will be added to one (this is control: finish the recursive call then come back to add 1)
- The control context grows with every function call (i.e., more promises to remember)

# Continuation-Passing Interpreters

- Consider:

```
(define (add a b)
  (define (add-iter b res)
    (if (= b 0)
        res
        (add-iter (- b 1) (+ res 1))))
  (add-iter b a))
```

# Continuation-Passing Interpreters

- Consider:

```
(define (add a b)
  (define (add-iter b res)
    (if (= b 0)
        res
        (add-iter (- b 1) (+ res 1))))
  (add-iter b a))
```

- Trace (add 3 3):

```
= (add-iter 3 3)
= (add-iter 2 4)
= (add-iter 1 5)
= (add-iter 0 6)
= 6
```

# Continuation-Passing Interpreters

- Consider:

```
(define (add a b)
  (define (add-iter b res)
    (if (= b 0)
        res
        (add-iter (- b 1) (+ res 1))))
  (add-iter b a))
```

- Trace (add 3 3):  
= (add-iter 3 3)  
= (add-iter 2 4)  
= (add-iter 1 5)  
= (add-iter 0 6)  
= 6
- Add iter is always invoked in the same control context
- Call at the tail-end means no promises to do anything (no need to return and do anything)
- No need to remember what to do with the result
- Only a constant amount of memory is needed regardless of the number of (recursive) calls: memory for a and for b

# Continuation-Passing Interpreters

- ```
(define (add a b)
  (if (= b 0)
      a
      (+ 1 (add a (- b 1))))))
```
- Function is called in an *operand position*
- Requires remembering the context to finish evaluating the outer call later (i.e., add the 1)

# Continuation-Passing Interpreters

- ```
(define (add a b)
  (if (= b 0)
      a
      (+ 1 (add a (- b 1))))))
```
- Function is called in an *operand position*
- Requires remembering the context to finish evaluating the outer call later (i.e., add the 1)
- **IT IS THE EVALUATION OF OPERANDS, NOT CALLING PROCEDURES, THAT REQUIRES THE CONTROL CONTEXT TO GROW**

# Continuation-Passing Interpreters

- Goal: Learn to track and manipulate control contexts





# Continuation-Passing Interpreters

Part V:  
Continuation-  
Passing  
Interpreters

Marco T.  
Morazán

Continuation-  
Passing  
Interpreter

A  
Trampolined  
Interpreter

An Imperative  
Interpreter

Exceptions

Threads

- Goal: Learn to track and manipulate control contexts
- Continuations
  - An abstraction for the control context notion
  - Our interpreter will explicitly pass a continuation (i.e., do this when operand is evaluated)
  - Represents a procedure that takes the result of an operand expression and completes the computation
  - A continuation is a function!
  - We will need the ability, therefore, to apply continuations to values

# Continuation-Passing Interpreters

Part V:  
Continuation-  
Passing  
Interpreters

Marco T.  
Morazán

Continuation-  
Passing  
Interpreter

A  
Trampolined  
Interpreter

An Imperative  
Interpreter

Exceptions

Threads

- Goal: Learn to track and manipulate control contexts
- Continuations
  - An abstraction for the control context notion
  - Our interpreter will explicitly pass a continuation (i.e., do this when operand is evaluated)
  - Represents a procedure that takes the result of an operand expression and completes the computation
  - A continuation is a function!
  - We will need the ability, therefore, to apply continuations to values
- ```
;; apply-cont: continuation value → expval  
;; Purpose: To apply the given continuation to the  
;;           given value and return the final answer
```
- A value may be anything that is computed by the program or that is computed to evaluate the program

# Continuation-Passing Interpreters

- Goal: Learn to track and manipulate control contexts
- Continuations
  - An abstraction for the control context notion
  - Our interpreter will explicitly pass a continuation (i.e., do this when operand is evaluated)
  - Represents a procedure that takes the result of an operand expression and completes the computation
  - A continuation is a function!
  - We will need the ability, therefore, to apply continuations to values
- `;; apply-cont: continuation value → expval`  
`;; Purpose: To apply the given continuation to the`  
`;;            given value and return the final answer`
- A value may be anything that is computed by the program or that is computed to evaluate the program
- We shall discover the needed continuation constructors as we analyze the LETREC language interpreter

# Continuation-Passing Interpreters

Part V:  
Continuation-  
Passing  
Interpreters

Marco T.  
Morazán

Continuation-  
Passing  
Interpreter

A  
Trampoline  
Interpreter

An Imperative  
Interpreter

Exceptions

Threads

- `;; value-of-program : program → expval`  
`(define (value-of-program pgm)`  
  `(cases program pgm`  
    `(a-program (exp1)`  
      `(value-of/k exp1 (init-env))))))`
- The value of `exp1` must be evaluated. **What needs to be done with its value?**

# Continuation-Passing Interpreters

Part V:  
Continuation-  
Passing  
Interpreters

Marco T.  
Morazán

Continuation-  
Passing  
Interpreter

A  
Trampoline  
Interpreter

An Imperative  
Interpreter

Exceptions

Threads

- `;; value-of-program : program → expval`  
`(define (value-of-program pgm)`  
  `(cases program pgm`  
    `(a-program (exp1)`  
      `(value-of/k exp1 (init-env))))))`
- The value of `exp1` must be evaluated. **What needs to be done with its value?**
- We need a continuation that simply returns the value it is applied to

# Continuation-Passing Interpreters

- `;; value-of-program : program → expval`  
`(define (value-of-program pgm)`  
  `(cases program pgm`  
    `(a-program (exp1)`  
      `(value-of/k exp1 (init-env))))))`
- The value of `exp1` must be evaluated. **What needs to be done with its value?**
- We need a continuation that simply returns the value it is applied to
- `;; value-of-program : program → expval`  
`(define (value-of-program pgm)`  
  `(cases program pgm`  
    `(a-program (exp1)`  
      `(value-of/k exp1 (init-env) (end-cont))))))`
- Semantics:  
  `(apply-cont (end-cont) val)`  
  `= (begin (display "End of computation.\n") val)`

# Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  
`(define (value-of exp env)`  
    `(cases expression exp`  
  
        `(const-exp (num) (num-val num))`  
  
    )
- The computation ends by returning a value
- In continuation-passing style, the continuation must finish the computation

# Continuation-Passing Interpreters

Part V:  
Continuation-  
Passing  
Interpreters

Marco T.  
Morazán

Continuation-  
Passing  
Interpreter

A  
Trampoline  
Interpreter

An Imperative  
Interpreter

Exceptions

Threads

- `;; value-of : expression environment  $\rightarrow$  expval`  
`(define (value-of exp env)`  
    `(cases expression exp`  
  
        `(const-exp (num) (num-val num))`  
  
• The computation ends by returning a value
- In continuation-passing style, the continuation must finish the computation
- `;; expression environment continuation  $\rightarrow$  expval`  
`(define (value-of/k exp env k)`  
    `(cases expression exp`  
  
        `(const-exp (num) (apply-cont k (num-val num)))`



# Continuation-Passing Interpreters

Part V:  
Continuation-  
Passing  
Interpreters

Marco T.  
Morazán

Continuation-  
Passing  
Interpreter

A  
Trampoline  
Interpreter

An Imperative  
Interpreter

Exceptions

Threads

- ```
;; value-of : expression environment → expval  
(define (value-of exp env)  
  (cases expression exp  
  
    (true-exp () (bool-val #t))  
  
    (false-exp () (bool-val #f))
```
- The computation ends by returning a value
- In continuation-passing style, the continuation must finish the computation

# Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  
`(define (value-of exp env)`  
    `(cases expression exp`  
  
        `(true-exp () (bool-val #t))`  
  
        `(false-exp () (bool-val #f))`
- The computation ends by returning a value
- In continuation-passing style, the continuation must finish the computation
- `;; expression environment continuation → expval`  
`(define (value-of/k exp env k)`  
    `(cases expression exp`  
  
        `(true-exp () (apply-cont k (bool-val #t)))`  
  
        `(false-exp () (apply-cont k (bool-val #f)))`

# Continuation-Passing Interpreters

Part V:  
Continuation-  
Passing  
Interpreters

Marco T.  
Morazán

Continuation-  
Passing  
Interpreter

A  
Trampoline  
Interpreter

An Imperative  
Interpreter

Exceptions

Threads

- `;; value-of : expression environment → expval`  
`(define (value-of exp env)`  
    `(cases expression exp`  
  
        `(var-exp (var) (apply-env env var))`  
- The computation ends by returning a value
- In continuation-passing style, the continuation must finish the computation

# Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  
`(define (value-of exp env)`  
    `(cases expression exp`  
  
        `(var-exp (var) (apply-env env var))`
- The computation ends by returning a value
- In continuation-passing style, the continuation must finish the computation
- `;; expression environment continuation → expval`  
`(define (value-of/k exp env k)`  
    `(cases expression exp`  
  
        `(var-exp (var) (apply-cont k (apply-env env var)))`

# Continuation-Passing Interpreters

Part V:  
Continuation-  
Passing  
Interpreters

Marco T.  
Morazán

Continuation-  
Passing  
Interpreter

A  
Trampoline  
Interpreter

An Imperative  
Interpreter

Exceptions

Threads

- `;; value-of : expression environment → expval`  
`(define (value-of exp env)`  
    `(cases expression exp`  
        `(proc-exp (params body)`  
            `(proc-val (procedure params body (vector env))))`  
    `)`
- The computation ends by returning a value
- In continuation-passing style, the continuation must finish the computation

# Continuation-Passing Interpreters

- `;; value-of : expression environment  $\rightarrow$  expval`  
`(define (value-of exp env)`  
    `(cases expression exp`  
        `(proc-exp (params body)`  
            `(proc-val (procedure params body (vector env))))`  
    )
- The computation ends by returning a value
- In continuation-passing style, the continuation must finish the computation
- `;; expression environment continuation  $\rightarrow$  expval`  
`(define (value-of/k exp env k)`  
    `(cases expression exp`  
        `(proc-exp (params body)`  
            `(apply-cont k (proc-val (procedure params body (vector env))))`  
    )

## Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  

```
(define (value-of exp env)
  (cases expression exp
    (zero?-exp (exp1)
      (let ((val1 (expval2num (value-of exp1 env))))
        (if (zero? val1)
            (bool-val #t)
            (bool-val #f)))))
```
- There is one non-tail call: evaluate `exp1`
- In continuation-passing style, the continuation must finish the computation

## Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  

```
(define (value-of exp env)
  (cases expression exp

    (zero?-exp (exp1)
      (let ((val1 (expval2num (value-of exp1 env))))
        (if (zero? val1)
            (bool-val #t)
            (bool-val #f)))))
```
- There is one non-tail call: evaluate `exp1`
- In continuation-passing style, the continuation must finish the computation
- `;; expression environment continuation → expval`  

```
(define (value-of/k exp env k)
  (cases expression exp

    (zero?-exp (exp1)
      (value-of/k exp1 env (zero?-cont k)))
```
- Save the given continuation, `k`, to use after value of `exp1` is known to finish the computation



## Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  
`(define (value-of exp env)`  
    `(cases expression exp`  
        `(zero?-exp (exp1)`  
            `(let ((val1 (expval2num (value-of exp1 env))))`  
                `(if (zero? val1)`  
                    `(bool-val #t)`  
                    `(bool-val #f))))`
- There is one non-tail call: evaluate exp1
- In continuation-passing style, the continuation must finish the computation
- `;; expression environment continuation → expval`  
`(define (value-of/k exp env k)`  
    `(cases expression exp`  
        `(zero?-exp (exp1)`  
            `(value-of/k exp1 env (zero?-cont k)))`
- Save the given continuation, k, to use after value of exp1 is known to finish the computation
- Semantics  
    `(apply-cont (zero?-cont cont) val)`  
    `= (if (zero? (expval2num val))`  
        `(apply-cont cont (bool-val #t))`  
        `(apply-cont cont (bool-val #f)))`

## Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  

```
(define (value-of exp env)  
  (cases expression exp  
  
    (if-exp (exp1 exp2 exp3)  
      (let ((val1 (value-of exp1 env)))  
        (if (expval2bool val1)  
            (value-of exp2 env)  
            (value-of exp3 env))))))
```
- There is one non-tail call: evaluate `exp1`
- To finish the computation `exp2`, `exp3`, and `env` must be remembered
- Since a concrete value is not returned, the continuation must be remembered to be used after evaluating `exp2` or `exp3`

## Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  

```
(define (value-of exp env)  
  (cases expression exp  
  
    (if-exp (exp1 exp2 exp3)  
      (let ((val1 (value-of exp1 env)))  
        (if (expval2bool val1)  
            (value-of exp2 env)  
            (value-of exp3 env))))))
```
- There is one non-tail call: evaluate `exp1`
- To finish the computation `exp2`, `exp3`, and `env` must be remembered
- Since a concrete value is not returned, the continuation must be remembered to be used after evaluating `exp2` or `exp3`
- `;; expression environment continuation → expval`  

```
(define (value-of/k exp env k)  
  (cases expression exp  
  
    (if-exp (exp1 exp2 exp3)  
      (value-of/k exp1 env (if-cont exp2 exp3 env k))))
```

## Continuation-Passing Interpreters

- `;; value-of : expression environment  $\rightarrow$  expval`  
`(define (value-of exp env)`  
    `(cases expression exp`  
        `(if-exp (exp1 exp2 exp3)`  
            `(let ((val1 (value-of exp1 env)))`  
                `(if (expval2bool val1)`  
                    `(value-of exp2 env)`  
                    `(value-of exp3 env))))))`
- There is one non-tail call: evaluate `exp1`
- To finish the computation `exp2`, `exp3`, and `env` must be remembered
- Since a concrete value is not returned, the continuation must be remembered to be used after evaluating `exp2` or `exp3`
- `;; expression environment continuation  $\rightarrow$  expval`  
`(define (value-of/k exp env k)`  
    `(cases expression exp`  
        `(if-exp (exp1 exp2 exp3)`  
            `(value-of/k exp1 env (if-cont exp2 exp3 env k))))`
- Semantics  
    `(apply-cont (if-cont (exp2 exp3 env saved-cont) val)`  
    `= (if (expval2bool val)`  
        `(value-of/k exp2 env saved-cont)`  
        `(value-of/k exp3 env saved-cont)))`

# Continuation-Passing Interpreters

Part V:  
Continuation-  
Passing  
Interpreters

Marco T.  
Morazán

Continuation-  
Passing  
Interpreter

A  
Trampoline  
Interpreter

An Imperative  
Interpreter

Exceptions

Threads

- `;; value-of : expression environment → expval`  
`(define (value-of exp env)`  
    `(cases expression exp`  
  
        `(letrec-exp (names params bodies letrec-body)`  
            `(value-of letrec-body (mk-letrec-env names params bodies env))))`
- There is one non-tail call: Create a letrec-env
- To finish the computation letrec-body must be remembered to evaluate it after the environment is computed
- Since a concrete value is not returned, the continuation must be remembered to be used after evaluating letrec-body

# Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  
`(define (value-of exp env)`  
  `(cases expression exp`  
  
    `(letrec-exp (names params bodies letrec-body)`  
      `(value-of letrec-body (mk-letrec-env names params bodies env))))`
- There is one non-tail call: Create a `letrec-env`
- To finish the computation `letrec-body` must be remembered to evaluate it after the environment is computed
- Since a concrete value is not returned, the continuation must be remembered to be used after evaluating `letrec-body`
- `;; expression environment continuation → expval`  
`(define (value-of/k exp env k)`  
  `(cases expression exp`  
  
    `(letrec-exp (names params bodies letrec-body)`  
      `(mk-letrec-env/k names params bodies env (letrec-cont letrec-body`

# Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  
`(define (value-of exp env)`  
  `(cases expression exp`  
  
    `(letrec-exp (names params bodies letrec-body)`  
      `(value-of letrec-body (mk-letrec-env names params bodies env))))`
- There is one non-tail call: Create a `letrec-env`
- To finish the computation `letrec-body` must be remembered to evaluate it after the environment is computed
- Since a concrete value is not returned, the continuation must be remembered to be used after evaluating `letrec-body`
- `;; expression environment continuation → expval`  
`(define (value-of/k exp env k)`  
  `(cases expression exp`  
  
    `(letrec-exp (names params bodies letrec-body)`  
      `(mk-letrec-env/k names params bodies env (letrec-cont letrec-body`
- Semantics  
  
  `(apply-cont (letrec-cont letrec-body saved-cont) val)`  
  `= (value-of/k letrec-body val saved-cont))`

## Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  

```
(define (value-of exp env)  
  (cases expression exp  
    (let-exp (vars exps body)  
      (let [(vals (map (lambda (e) (value-of e env)) exps))]  
        (value-of body (foldr (lambda (var val acc)  
                                (extend-env var val acc))  
                              env vars vals))))))
```
- There are two non-tail calls: `map` and `foldr`
- To finish the computation body, `vars`, and `env` must be remembered
- The continuation remembered to use after evaluating body



## Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  

```
(define (value-of exp env)  
  (cases expression exp  
    (let-exp (vars exps body)  
      (let [(vals (map (lambda (e) (value-of e env)) exps))]  
        (value-of body (foldr (lambda (var val acc)  
                                (extend-env var val acc))  
                              env vars vals))))))
```
- There are two non-tail calls: `map` and `foldr`
- To finish the computation body, vars, and env must be remembered
- The continuation remembered to use after evaluating body
- `;; expression environment continuation → expval`  

```
(define (value-of/k exp env k)  
  (cases expression exp  
    (let-exp (vars exps body)  
      (eval-rands/k exps env (let1-cont vars body env k))))
```
- Save the given continuation, k, to use after value of body is known to finish the computation

## Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  

```
(define (value-of exp env)  
  (cases expression exp  
    (let-exp (vars exps body)  
      (let [(vals (map (lambda (e) (value-of e env)) exps))]  
        (value-of body (foldr (lambda (var val acc)  
                                (extend-env var val acc))  
                              env vars vals))))))
```
- There are two non-tail calls: `map` and `foldr`
- To finish the computation body, `vars`, and `env` must be remembered
- The continuation remembered to use after evaluating body
- `;; expression environment continuation → expval`  

```
(define (value-of/k exp env k)  
  (cases expression exp  
    (let-exp (vars exps body)  
      (eval-rands/k exps env (let1-cont vars body env k))))
```
- Save the given continuation, `k`, to use after value of body is known to finish the computation
- Semantics  

```
(apply-cont (let1-cont (vars body env saved-cont)) val)  
= (create-let-lenv vars val env (let2-cont body saved-cont))  
  
(apply-cont (let2-cont (body saved-cont)) val)  
= (value-of/k body val saved-cont)
```

## Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  

```
(define (value-of exp env)  
  (cases expression exp  
  
    (diff-exp (exp1 exp2)  
              (let ((num1 (expval2num (value-of exp1 env)))  
                    (num2 (expval2num (value-of exp2 env))))  
                (num-val (- num1 num2)))))
```
- There are two non-tail function calls evaluate exp1 and evaluate exp2
- To finish the computation after evaluating exp1, the value of exp2 and env must be remembered
- Since a concrete value is not returned, the continuation must be remembered to be used after evaluating exp1

## Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  

```
(define (value-of exp env)  
  (cases expression exp  
  
    (diff-exp (exp1 exp2)  
              (let ((num1 (expval2num (value-of exp1 env)))  
                    (num2 (expval2num (value-of exp2 env))))  
                (num-val (- num1 num2))))  
  
    (value-of/k exp1 env (diff-cont1 exp2 env k)))
```
- There are two non-tail function calls evaluate exp1 and evaluate exp2
- To finish the computation after evaluating exp1, the value of exp2 and env must be remembered
- Since a concrete value is not returned, the continuation must be remembered to be used after evaluating exp1
- A continuation is needed to finish the computation when exp2 is evaluated
- Must remember the value of exp1 and the continuation to use after exp2 is evaluated

## Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  

```
(define (value-of exp env)  
  (cases expression exp  
  
    (diff-exp (exp1 exp2)  
              (let ((num1 (expval2num (value-of exp1 env)))  
                    (num2 (expval2num (value-of exp2 env))))  
                (num-val (- num1 num2))))  
  
    (value-of/k exp1 env (diff-cont1 exp2 env k)))  
  
    (diff-cont1 (exp2 env saved-cont)  
               = (value-of/k exp2 env (diff-cont2 val saved-cont)))  
  
               (diff-cont2 (val1 saved-cont)  
                           = (apply-cont saved-cont (num-val (- (expval2num val1)  
                                                                    (expval2num val))))))
```
- There are two non-tail function calls evaluate exp1 and evaluate exp2
- To finish the computation after evaluating exp1, the value of exp2 and env must be remembered
- Since a concrete value is not returned, the continuation must be remembered to be used after evaluating exp1
- A continuation is needed to finish the computation when exp2 is evaluated
- Must remember the value of exp1 and the continuation to use after exp2 is evaluated
- Semantics

## Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  

```
(define (value-of exp env)  
  (cases expression exp  
    (call-exp (rator rands)  
              (let [(proc (expval2proc (value-of rator env))]  
                    (args (map (lambda (rand) (value-of rand env))  
                               (apply-procedure proc args)))))
```
- There are two non-tail function calls:
  - Evaluate rator
  - Evaluate rands
- To finish the computation after evaluating rator, the value of rands and env must be remembered
- Since a concrete value is not returned, the continuation must be remembered to be used after evaluating rator

## Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  

```
(define (value-of exp env)  
  (cases expression exp  
    (call-exp (rator rands)  
              (let [(proc (expval2proc (value-of rator env))]  
                    (args (map (lambda (rand) (value-of rand env))  
                               (apply-procedure proc args)))))
```
- There are two non-tail function calls:
  - Evaluate rator
  - Evaluate rands
- To finish the computation after evaluating rator, the value of rands and env must be remembered
- Since a concrete value is not returned, the continuation must be remembered to be used after evaluating rator
  - ```
(rator-cont (rands env saved-cont)  
            (eval-rands/k rands env (rands-cont val saved-cont)))
```
- The continuation is needed to finish computation after rands are evaluated
- Must remember rator value to use after rands is evaluated

## Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  

```
(define (value-of exp env)  
  (cases expression exp  
    (call-exp (rator rands)  
              (let [(proc (expval2proc (value-of rator env)))  
                    (args (map (lambda (rand) (value-of rand env))  
                               (apply-procedure proc args)))
```
- There are two non-tail function calls:
  - Evaluate rator
  - Evaluate rands
- To finish the computation after evaluating rator, the value of rands and env must be remembered
- Since a concrete value is not returned, the continuation must be remembered to be used after evaluating rator
  - ```
(rator-cont (rands env saved-cont)  
            (eval-rands/k rands env (rands-cont val saved-cont)))
```
- The continuation is needed to finish computation after rands are evaluated
- Must remember rator value to use after rands is evaluated



## Continuation-Passing Interpreters

- `;; value-of : expression environment → expval`  

```
(define (value-of exp env)  
  (cases expression exp  
    (call-exp (rator rands)  
              (let [(proc (expval2proc (value-of rator env)))  
                    (args (map (lambda (rand) (value-of rand env))  
                               (apply-procedure proc args)))]
```
- There are two non-tail function calls:
  - Evaluate rator
  - Evaluate rands
- To finish the computation after evaluating rator, the value of rands and env must be remembered
- Since a concrete value is not returned, the continuation must be remembered to be used after evaluating rator
- ```
(rator-cont (rands env saved-cont)  
            (eval-rands/k rands env (rands-cont val saved-cont)))
```
- The continuation is needed to finish computation after rands are evaluated
- Must remember rator value to use after rands is evaluated
- Semantics  

```
(rator-cont (rands env saved-cont)  
= (eval-rands/k rands env (rands-cont val saved-cont)))  
  
(rands-cont (rator saved-cont)  
= (apply-procedure/k (expval2proc rator) val saved-cont))
```

# Continuation-Passing Interpreters

- ```
;; proc (listof expval) continuation → expval
;; Purpose: Apply the given procedure to the given values
(define (apply-procedure/k f vals k)
  (cases proc f
    (procedure (params body envv)
      (let [(saved-env (vector-ref envv 0))]
        (value-of/k body
                     (foldr (lambda (binding acc)
                             (extend-env (car binding)
                                           (cadr binding)
                                           acc))
                           saved-env
                           (map (lambda (p v) (list p v))
                               params
                               vals))
                     k))))))
```
- Continuation input for call to value-of
- Create new env to evaluate the body assuming foldr and map do not grow the control context

# Continuation-Passing Interpreters

- ```
;; (listof symbol) (listof (listof symbol)) (listof expression)
;; environment      continuation
;; → environment
;; Purpose: Add the proc-vals for the given procedures in the given environment
(define (mk-letrec-env/k names params bodies env k)
  (let* [(temp-proc-vals (map (lambda (p b)
                                (proc-val (procedure p b (vector (empty-env))))))
        (params
         bodies))
        (new-env (foldl (lambda (name proc env) (extend-env name proc env))
                        env
                        names
                        temp-proc-vals))]
    (begin
      (for-each (lambda (p)
                  (cases proc p
                    (procedure (p b ve) (vector-set! ve 0 new-env))))
                (map (lambda (p) (expval2proc p)) temp-proc-vals))
      (apply-cont k new-env))))
```
- Temporarily create incorrect proc-vals for the locally defined recursive functions assuming map does not grow the control context.

# Continuation-Passing Interpreters

- ```
;; (listof symbol) (listof (listof symbol)) (listof expression)
;; environment      continuation
;; → environment
;; Purpose: Add the proc-vals for the given procedures in the given environment
(define (mk-letrec-env/k names params bodies env k)
  (let* [(temp-proc-vals (map (lambda (p b)
                                (proc-val (procedure p b (vector (empty-env))))))
        (params
         bodies))
        (new-env (foldl (lambda (name proc env) (extend-env name proc env))
                        env
                        names
                        temp-proc-vals))]
    (begin
      (for-each (lambda (p)
                  (cases proc p
                    (procedure (p b ve) (vector-set! ve 0 new-env))))
                (map (lambda (p) (expval2proc p)) temp-proc-vals))
      (apply-cont k new-env))))
```
- Temporarily create incorrect proc-vals for the locally defined recursive functions assuming map does not grow the control context.
- Create new env by adding incorrect procs to the given environment

# Continuation-Passing Interpreters

- ```
;; (listof symbol) (listof (listof symbol)) (listof expression)
;; environment      continuation
;; → environment
;; Purpose: Add the proc-vals for the given procedures in the given environment
(define (mk-letrec-env/k names params bodies env k)
  (let* [(temp-proc-vals (map (lambda (p b)
                                (proc-val (procedure p b (vector (empty-env))))))
        params
        bodies))
    (new-env (foldl (lambda (name proc env) (extend-env name proc env))
                  env
                  names
                  temp-proc-vals))]
    (begin
      (for-each (lambda (p)
                  (cases proc p
                    (procedure (p b ve) (vector-set! ve 0 new-env))))
                (map (lambda (p) (expval2proc p)) temp-proc-vals))
      (apply-cont k new-env))))
```
- Temporarily create incorrect proc-vals for the locally defined recursive functions assuming map does not grow the control context.
- Create a new env by adding incorrect procs to the given environment
- Correct the environment in each incorrect proc

# Continuation-Passing Interpreters

- ```
;; (listof symbol) (listof (listof symbol)) (listof expression)
;; environment      continuation
;; → environment
;; Purpose: Add the proc-vals for the given procedures in the given environment
(define (mk-letrec-env/k names params bodies env k)
  (let* [(temp-proc-vals (map (lambda (p b)
                                (proc-val (procedure p b (vector (empty-env))))))
        params
        bodies))
    (new-env (foldl (lambda (name proc env) (extend-env name proc env))
                  env
                  names
                  temp-proc-vals))]
      (begin
        (for-each (lambda (p)
                    (cases proc p
                      (procedure (p b ve) (vector-set! ve 0 new-env))))
                  (map (lambda (p) (expval2proc p)) temp-proc-vals))
        (apply-cont k new-env))))
```
- Temporarily create incorrect proc-vals for the locally defined recursive functions assuming map does not grow the control context.
- Create a new env by adding incorrect procs to the given environment
- Correct the environment in each incorrect proc
- Finish the computation by applying the continuation to the correct new env

# Continuation-Passing Interpreters

- ```
;; (listof expression) environment continuation → expval
;; Purpose: Evaluate the given list of exprs and apply the
;;           given cont
(define (eval-rands/k rands env k)
  (if (null? rands)
      (apply-cont k '())
      (value-of/k (car rands)
                  env
                  (eval-rands-cont1 (cdr rands) env k))))
```
- Two steps: evaluate the first argument and evaluate the rest of the arguments

# Continuation-Passing Interpreters

- ```
;; (listof expression) environment continuation → expval
;; Purpose: Evaluate the given list of exprs and apply the
;;           given cont
(define (eval-rands/k rands env k)
  (if (null? rands)
      (apply-cont k '())
      (value-of/k (car rands)
                  env
                  (eval-rands-cont1 (cdr rands) env k))))
```
- Two steps: evaluate the first argument and evaluate the rest of the arguments
- Evaluate the first argument using a continuation that evaluates the rest of the arguments



# Continuation-Passing Interpreters

- ```
;; (listof expression) environment continuation → expval  
;; Purpose: Evaluate the given list of exprs and apply the  
;;           given cont  
(define (eval-rands/k rands env k)  
  (if (null? rands)  
      (apply-cont k '())  
      (value-of/k (car rands)  
                  env  
                  (eval-rands-cont1 (cdr rands) env k))))
```
- Two steps: evaluate the first argument and evaluate the rest of the arguments
- Evaluate the first argument using a continuation that evaluates the rest of the arguments
- If there are no more arguments to evaluate apply the given continuation to the empty list of evaluated arguments

# Continuation-Passing Interpreters

- ```
;; (listof symbol) (listof expval) environment continuation  
;; → expval  
;; Purpose: Apply the given cont to the env created using the  
;;      given variables, expvals, and environment  
;; ACC INVARIANT  
;; env contains the processed bindings  
(define (create-let-env vars vals env k)  
  (if (empty? vars)  
      (apply-cont k env)  
      (create-let-env (cdr vars)  
                      (cdr vals)  
                      (extend-env (car vars) (car vals) env)  
                      k))))
```
- Accumulate new bindings in the given environment

# Continuation-Passing Interpreters

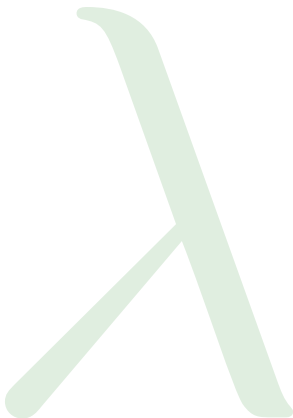
- ```
;; (listof symbol) (listof expval) environment continuation  
;; → expval  
;; Purpose: Apply the given cont to the env created using the  
;;      given variables, expvals, and environment  
;; ACC INVARIANT  
;; env contains the processed bindings  
(define (create-let-lenv vars vals env k)  
  (if (empty? vars)  
      (apply-cont k env)  
      (create-let-lenv (cdr vars)  
                        (cdr vals)  
                        (extend-env (car vars) (car vals) env)  
                        k))))
```
- Accumulate new bindings in the given environment
- If no more bindings to add, apply continuation to the given environment that stores all the bindings

# Continuation-Passing Interpreters

- ```
;; (listof symbol) (listof expval) environment continuation  
;; → expval  
;; Purpose: Apply the given cont to the env created using the  
;;      given variables, expvals, and environment  
;; ACC INVARIANT  
;; env contains the processed bindings  
(define (create-let-lenv vars vals env k)  
  (if (empty? vars)  
      (apply-cont k env)  
      (create-let-lenv (cdr vars)  
                       (cdr vals)  
                       (extend-env (car vars) (car vals) env)  
                       k))))
```
- Accumulate new bindings in the given environment
- If no more bindings to add, apply continuation to the given environment that stores all the bindings
- No new continuations are needed: all function calls in tail-position

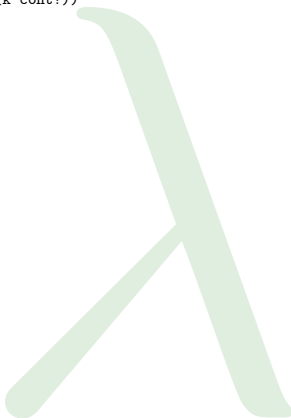
# Continuation-Passing Interpreters

- `(define-datatype continuation cont?  
 (end-cont))`



# Continuation-Passing Interpreters

- `(define-datatype continuation cont?  
 (end-cont))`
- `(zero?-cont (k cont?))`



# Continuation-Passing Interpreters

- `(define-datatype continuation cont?  
 (end-cont))`
- `(zero?-cont (k cont?))`
- `(letrec-cont (letrec-body expression?)  
 (saved-cont cont?))`



# Continuation-Passing Interpreters

- `(define-datatype continuation cont?  
 (end-cont))`
- `(zero?-cont (k cont?))`
- `(letrec-cont (letrec-body expression?)  
 (saved-cont cont?))`
- `(let1-cont (vrs (list-of symbol?))  
 (b expression?)  
 (e environment?)  
 (k cont?))  
 (let2-cont (b expression?)  
 (k cont?))`



# Continuation-Passing Interpreters

- `(define-datatype continuation cont?  
 (end-cont))`
- `(zero?-cont (k cont?))`
- `(letrec-cont (letrec-body expression?)  
 (saved-cont cont?))`
- `(let1-cont (vrs (list-of symbol?))  
 (b expression?)  
 (e environment?)  
 (k cont?))  
 (let2-cont (b expression?)  
 (k cont?))`
- `(if-cont (e2 expression?)  
 (e3 expression?)  
 (e environment?)  
 (k cont?))`

# Continuation-Passing Interpreters

- `(define-datatype continuation cont?  
 (end-cont))`
- `(zero?-cont (k cont?))`
- `(letrec-cont (letrec-body expression?  
 (saved-cont cont?))`
- `(let1-cont (vrs (list-of symbol?))  
 (b expression?)  
 (e environment?)  
 (k cont?))  
 (let2-cont (b expression?)  
 (k cont?))`
- `(if-cont (e2 expression?)  
 (e3 expression?)  
 (e environment?)  
 (k cont?))`
- `(diff-cont1 (e2 expression?)  
 (e environment?)  
 (k cont?))  
 (diff-cont2 (v1 expval?)  
 (k cont?))`

# Continuation-Passing Interpreters

- `(define-datatype continuation cont?  
 (end-cont))`
- `(zero?-cont (k cont?))`
- `(letrec-cont (letrec-body expression?)  
 (saved-cont cont?))`
- `(let1-cont (vrs (list-of symbol?))  
 (b expression?)  
 (e environment?)  
 (k cont?))  
  
 (let2-cont (b expression?)  
 (k cont?))`
- `(if-cont (e2 expression?)  
 (e3 expression?)  
 (e environment?)  
 (k cont?))`
- `(diff-cont1 (e2 expression?)  
 (e environment?)  
 (k cont?))  
  
 (diff-cont2 (v1 expval?)  
 (k cont?))`
- `(rator-cont (rnds (list-of expression?))  
 (e environment?)  
 (k cont?))  
  
 (rands-cont (operator expval?)  
 (k cont?))`

# Continuation-Passing Interpreters

- `(define-datatype continuation cont?  
 (end-cont))`
- `(zero?-cont (k cont?))`
- `(letrec-cont (letrec-body expression?)  
 (saved-cont cont?))`
- `(let1-cont (vrs (list-of symbol?))  
 (b expression?)  
 (e environment?)  
 (k cont?))  
  
 (let2-cont (b expression?)  
 (k cont?))`
- `(if-cont (e2 expression?)  
 (e3 expression?)  
 (e environment?)  
 (k cont?))`
- `(diff-cont1 (e2 expression?)  
 (e environment?)  
 (k cont?))  
  
 (diff-cont2 (v1 expval?)  
 (k cont?))`
- `(rator-cont (rnds (list-of expression?))  
 (e environment?)  
 (k cont?))  
  
 (rands-cont (operator expval?)  
 (k cont?))`
- `(eval-rands-cont1 (rands (list-of expression?))  
 (e environment?)  
 (k cont?))  
  
 (eval-rands-cont2 (farg expval?)  
 (k cont?)))`

# Continuation-Passing Interpreters

## Auxiliary Functions

- ```
;; continuation expval → expval
;; Purpose: Apply the given cont to the given value and return
;;           the final answer
(define (apply-cont k val)
  (cases continuation k
    (end-cont () val)
```

# Continuation-Passing Interpreters

## Auxiliary Functions

- ```
;; continuation expval → expval
;; Purpose: Apply the given cont to the given value and return
;;           the final answer
(define (apply-cont k val)
  (cases continuation k
    (end-cont () val)))
```
- ```
(zero?-cont (cont)
  (if (zero? (expval2num val))
      (apply-cont cont (bool-val #t))
      (apply-cont cont (bool-val #f)))))
```

# Continuation-Passing Interpreters

## Auxiliary Functions

- `;; continuation expval → expval`  
`;; Purpose: Apply the given cont to the given value and return`  
`;; the final answer`  
`(define (apply-cont k val)`  
 `(cases continuation k`  
 `(end-cont () val)`
- `(zero?-cont (cont)`  
 `(if (zero? (expval2num val))`  
 `(apply-cont cont (bool-val #t))`  
 `(apply-cont cont (bool-val #f))))`
- `(let1-cont (vars body env saved-cont)`  
 `(create-let-levn vars val env (let2-cont body saved-cont)))`

# Continuation-Passing Interpreters

## Auxiliary Functions

- `;; continuation expval → expval`  
`;; Purpose: Apply the given cont to the given value and return`  
`;; the final answer`  
`(define (apply-cont k val)`  
 `(cases continuation k`  
 `(end-cont () val)`
- `(zero?-cont (cont)`  
 `(if (zero? (expval2num val))`  
 `(apply-cont cont (bool-val #t))`  
 `(apply-cont cont (bool-val #f))))`
- `(let1-cont (vars body env saved-cont)`  
 `(create-let-levn vars val env (let2-cont body saved-cont)))`
- `(let2-cont (body saved-cont)`  
 `(value-of/k body val saved-cont))`



# Continuation-Passing Interpreters

## Auxiliary Functions

- `;; continuation expval → expval`  
`;; Purpose: Apply the given cont to the given value and return`  
`;; the final answer`  
`(define (apply-cont k val)`  
 `(cases continuation k`  
 `(end-cont () val)`
- `(zero?-cont (cont)`  
 `(if (zero? (expval2num val))`  
 `(apply-cont cont (bool-val #t))`  
 `(apply-cont cont (bool-val #f))))`
- `(let1-cont (vars body env saved-cont)`  
 `(create-let-letenv vars val env (let2-cont body saved-cont)))`
- `(let2-cont (body saved-cont)`  
 `(value-of/k body val saved-cont))`
- `(if-cont (exp2 exp3 env saved-cont)`  
 `(if (expval2bool val)`  
 `(value-of/k exp2 env saved-cont)`  
 `(value-of/k exp3 env saved-cont)))`

# Continuation-Passing Interpreters

## Auxiliary Functions

- ```
;; continuation expval → expval
;; Purpose: Apply the given cont to the given value and return the
(define (apply-cont k val)
  (cases continuation k
    (letrec-cont (letrec-body saved-cont)
      (value-of/k letrec-body val saved-cont)))
```

# Continuation-Passing Interpreters

## Auxiliary Functions

- `;; continuation expval → expval`  
`;; Purpose: Apply the given cont to the given value and return the`  
`(define (apply-cont k val)`  
 `(cases continuation k`  
 `(letrec-cont (letrec-body saved-cont)`  
 `(value-of/k letrec-body val saved-cont)))`
- `(diff-cont1 (exp2 env saved-cont)`  
 `(value-of/k exp2 env (diff-cont2 val saved-cont)))`

# Continuation-Passing Interpreters

## Auxiliary Functions

- `;; continuation expval → expval`  
`;; Purpose: Apply the given cont to the given value and return the`  
`(define (apply-cont k val)`  
 `(cases continuation k`  
 `(letrec-cont (letrec-body saved-cont)`  
 `(value-of/k letrec-body val saved-cont))`
- `(diff-cont1 (exp2 env saved-cont)`  
 `(value-of/k exp2 env (diff-cont2 val saved-cont)))`
- `(diff-cont2 (val1 saved-cont)`  
 `(apply-cont saved-cont (num-val (- (expval2num val1)`  
 `(expval2num val))))))`

# Continuation-Passing Interpreters

## Auxiliary Functions

- `;; continuation expval → expval`  
`;; Purpose: Apply the given cont to the given value and return the`  
`(define (apply-cont k val)`  
 `(cases continuation k`  
 `(letrec-cont (letrec-body saved-cont)`  
 `(value-of/k letrec-body val saved-cont))`
- `(diff-cont1 (exp2 env saved-cont)`  
 `(value-of/k exp2 env (diff-cont2 val saved-cont)))`
- `(diff-cont2 (val1 saved-cont)`  
 `(apply-cont saved-cont (num-val (- (expval2num val1)`  
 `(expval2num val))))))`
- `(rator-cont (rands env saved-cont)`  
 `(eval-rands/k rands env (rands-cont val saved-cont)))`

# Continuation-Passing Interpreters

## Auxiliary Functions

- `;; continuation expval → expval`  
`;; Purpose: Apply the given cont to the given value and return the`  
`(define (apply-cont k val)`  
 `(cases continuation k`  
 `(letrec-cont (letrec-body saved-cont)`  
 `(value-of/k letrec-body val saved-cont))`
- `(diff-cont1 (exp2 env saved-cont)`  
 `(value-of/k exp2 env (diff-cont2 val saved-cont)))`
- `(diff-cont2 (val1 saved-cont)`  
 `(apply-cont saved-cont (num-val (- (expval2num val1)`  
 `(expval2num val))))))`
- `(rator-cont (rands env saved-cont)`  
 `(eval-rands/k rands env (rands-cont val saved-cont)))`
- `(rands-cont (rator saved-cont)`  
 `(apply-procedure/k (expval2proc rator) val saved-cont))`

# Continuation-Passing Interpreters

## Auxiliary Functions

- `;; continuation expval → expval`  
`;; Purpose: Apply the given cont to the given value and return the`  
`(define (apply-cont k val)`  
 `(cases continuation k`  
 `(letrec-cont (letrec-body saved-cont)`  
 `(value-of/k letrec-body val saved-cont)))`
- `(diff-cont1 (exp2 env saved-cont)`  
 `(value-of/k exp2 env (diff-cont2 val saved-cont)))`
- `(diff-cont2 (val1 saved-cont)`  
 `(apply-cont saved-cont (num-val (- (expval2num val1)`  
 `(expval2num val))))))`
- `(rator-cont (rands env saved-cont)`  
 `(eval-rands/k rands env (rands-cont val saved-cont)))`
- `(rands-cont (rator saved-cont)`  
 `(apply-procedure/k (expval2proc rator) val saved-cont))`
- `(eval-rands-cont1 (rands env saved-cont)`  
 `(eval-rands/k rands env (eval-rands-cont2 val saved-cont)))`

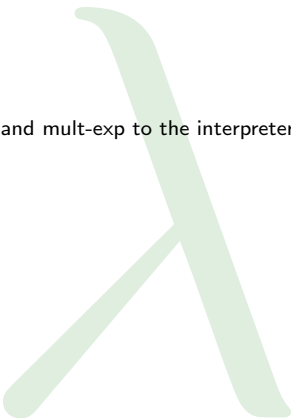
# Continuation-Passing Interpreters

## Auxiliary Functions

- `;; continuation expval → expval`  
`;; Purpose: Apply the given cont to the given value and return the`  
`(define (apply-cont k val)`  
 `(cases continuation k`  
 `(letrec-cont (letrec-body saved-cont)`  
 `(value-of/k letrec-body val saved-cont)))`
- `(diff-cont1 (exp2 env saved-cont)`  
 `(value-of/k exp2 env (diff-cont2 val saved-cont)))`
- `(diff-cont2 (val1 saved-cont)`  
 `(apply-cont saved-cont (num-val (- (expval2num val1)`  
 `(expval2num val))))))`
- `(rator-cont (rands env saved-cont)`  
 `(eval-rands/k rands env (rands-cont val saved-cont)))`
- `(rands-cont (rator saved-cont)`  
 `(apply-procedure/k (expval2proc rator) val saved-cont))`
- `(eval-rands-cont1 (rands env saved-cont)`  
 `(eval-rands/k rands env (eval-rands-cont2 val saved-cont)))`
- `(eval-rands-cont2 (first-rand k)`  
 `(apply-cont k (cons first-rand val))))`



- add sum-exp and mult-exp to the interpreter
- 5.4, **5.9\*\***



- We know will explore how to transform a continuation-passing interpreter to a *regular* PL
- By regular we mean no HOF
- Can use data structure representation of continuations



- We know will explore how to transform a continuation-passing interpreter to a *regular* PL
- By regular we mean no HOF
- Can use data structure representation of continuations
- More problematic:
  - Most PLs always grow the control context with every procedure call
  - Control context is usually a stack (that always grows with every procedure call)

- We know will explore how to transform a continuation-passing interpreter to a *regular* PL
- By regular we mean no HOF
- Can use data structure representation of continuations
- More problematic:
  - Most PLs always grow the control context with every procedure call
  - Control context is usually a stack (that always grows with every procedure call)
- Why do most PLs always grow the control context?

- We know will explore how to transform a continuation-passing interpreter to a *regular* PL
- By regular we mean no HOF
- Can use data structure representation of continuations
- More problematic:
  - Most PLs always grow the control context with every procedure call
  - Control context is usually a stack (that always grows with every procedure call)
- Why do most PLs always grow the control context?
  - Almost all procedure calls occur at the RHS of an assignment statement:  $x = f(\dots)$
  - The assignment is a delayed operation and requires growing the control context to track its pending execution
  - Environment info is also placed on the stack & removed

- One solution for such languages is trampolining
- Break an unbounded chain of proc calls by having one of the procedures in the interpreter return a zero-argument proc
- When called, this proc continues the computation
- The computation bounces from one procedure call to the next
- This is controlled by a trampoline procedure

- One solution for such languages is trampolining
- Break an unbounded chain of proc calls by having one of the procedures in the interpreter return a zero-argument proc
- When called, this proc continues the computation
- The computation bounces from one procedure call to the next
- This is controlled by a trampoline procedure
- What procedure calls?

- One solution for such languages is trampolining
- Break an unbounded chain of proc calls by having one of the procedures in the interpreter return a zero-argument proc
- When called, this proc continues the computation
- The computation bounces from one procedure call to the next
- This is controlled by a trampoline procedure
- What procedure calls?
- Those that are steps in the evaluator: `value-of/k`, `apply-cont`, `apply-procedure/k`, `eval-rands/k`, and `mk-letrec-env/k`
- Instead of calling one of them (and grow the control context), make calls to these functions the body of a zero-argument function that is given to the trampoline function



- Let illustrate how this works

```
(trampoline (value-of-program (value-of/k -(3, 1) (empty-env) (end-cont))))
```



- Let illustrate how this works  

```
(trampoline (value-of-program (value-of/k -(3, 1) (empty-env) (end-cont))))
```
- = (trampoline (lambda () (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont)))))

- Let illustrate how this works  

```
(trampoline (value-of-program (value-of/k -(3, 1) (empty-env) (end-cont))))
```
- = (trampoline (lambda () (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont)))))
- = (trampoline (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont))))

- Let illustrate how this works  

```
(trampoline (value-of-program (value-of/k -(3, 1) (empty-env) (end-cont))))
```
- = (trampoline (lambda () (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont)))))
- = (trampoline (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont))))
- = (trampoline (lambda () (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3)))

- Let illustrate how this works

```
(trampoline (value-of-program (value-of/k -(3, 1) (empty-env) (end-cont))))
```
- = (trampoline (lambda () (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont)))))
- = (trampoline (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont))))
- = (trampoline (lambda () (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3)))
- = (trampoline (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3))

- Let illustrate how this works

```
(trampoline (value-of-program (value-of/k -(3, 1) (empty-env) (end-cont))))
```
- = (trampoline (lambda () (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont)))))
- = (trampoline (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont))))
- = (trampoline (lambda () (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3)))
- = (trampoline (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3))
- = (trampoline (lambda () (value-of/k 1 (empty-env) (diff-cont2 3 (end-cont)))))

- Let illustrate how this works

```
(trampoline (value-of-program (value-of/k -(3, 1) (empty-env) (end-cont))))
```
- = (trampoline (lambda () (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont)))))
- = (trampoline (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont))))
- = (trampoline (lambda () (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3)))
- = (trampoline (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3))
- = (trampoline (lambda () (value-of/k 1 (empty-env) (diff-cont2 3 (end-cont)))))
- = (trampoline (value-of/k 1 (empty-env) (diff-cont2 3 (end-cont))))

- Let illustrate how this works

```
(trampoline (value-of-program (value-of/k -(3, 1) (empty-env) (end-cont))))
```
- = (trampoline (lambda () (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont)))))
- = (trampoline (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont))))
- = (trampoline (lambda () (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3)))
- = (trampoline (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3))
- = (trampoline (lambda () (value-of/k 1 (empty-env) (diff-cont2 3 (end-cont)))))
- = (trampoline (value-of/k 1 (empty-env) (diff-cont2 3 (end-cont))))
- = (trampoline (lambda () (apply-cont (diff-cont2 3 (end-cont)) 1)))



- Let illustrate how this works

```
(trampoline (value-of-program (value-of/k -(3, 1) (empty-env) (end-cont))))
```
- = (trampoline (lambda () (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont)))))
- = (trampoline (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont))))
- = (trampoline (lambda () (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3)))
- = (trampoline (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3))
- = (trampoline (lambda () (value-of/k 1 (empty-env) (diff-cont2 3 (end-cont)))))
- = (trampoline (value-of/k 1 (empty-env) (diff-cont2 3 (end-cont))))
- = (trampoline (lambda () (apply-cont (diff-cont2 3 (end-cont)) 1)))
- = (trampoline (apply-cont (diff-cont2 3 (end-cont)) 1))

- Let illustrate how this works

```
(trampoline (value-of-program (value-of/k -(3, 1) (empty-env) (end-cont))))  
• = (trampoline (lambda () (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont)))))  
• = (trampoline (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont))))  
• = (trampoline (lambda () (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3)))  
• = (trampoline (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3))  
• = (trampoline (lambda () (value-of/k 1 (empty-env) (diff-cont2 3 (end-cont)))))  
• = (trampoline (value-of/k 1 (empty-env) (diff-cont2 3 (end-cont))))  
• = (trampoline (lambda () (apply-cont (diff-cont2 3 (end-cont)) 1)))  
• = (trampoline (apply-cont (diff-cont2 3 (end-cont)) 1))  
• = (trampoline (lambda () (apply-cont (end-cont) 2)))
```

- Let illustrate how this works

```
(trampoline (value-of-program (value-of/k -(3, 1) (empty-env) (end-cont))))
```
- = (trampoline (lambda () (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont)))))
- = (trampoline (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont))))
- = (trampoline (lambda () (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3)))
- = (trampoline (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3))
- = (trampoline (lambda () (value-of/k 1 (empty-env) (diff-cont2 3 (end-cont)))))
- = (trampoline (value-of/k 1 (empty-env) (diff-cont2 3 (end-cont))))
- = (trampoline (lambda () (apply-cont (diff-cont2 3 (end-cont)) 1)))
- = (trampoline (apply-cont (diff-cont2 3 (end-cont)) 1))
- = (trampoline (lambda () (apply-cont (end-cont) 2)))
- = (trampoline (apply-cont (end-cont) 2))

- Let illustrate how this works

```
(trampoline (value-of-program (value-of/k -(3, 1) (empty-env) (end-cont))))
```
- = (trampoline (lambda () (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont)))))
- = (trampoline (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont))))
- = (trampoline (lambda () (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3)))
- = (trampoline (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3))
- = (trampoline (lambda () (value-of/k 1 (empty-env) (diff-cont2 3 (end-cont)))))
- = (trampoline (value-of/k 1 (empty-env) (diff-cont2 3 (end-cont))))
- = (trampoline (lambda () (apply-cont (diff-cont2 3 (end-cont)) 1)))
- = (trampoline (apply-cont (diff-cont2 3 (end-cont)) 1))
- = (trampoline (lambda () (apply-cont (end-cont) 2)))
- = (trampoline (apply-cont (end-cont) 2))
- = (trampoline (lambda () 2))

- Let illustrate how this works

```
(trampoline (value-of-program (value-of/k -(3, 1) (empty-env) (end-cont))))
```
- = (trampoline (lambda () (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont)))))
- = (trampoline (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont))))
- = (trampoline (lambda () (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3)))
- = (trampoline (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3))
- = (trampoline (lambda () (value-of/k 1 (empty-env) (diff-cont2 3 (end-cont)))))
- = (trampoline (value-of/k 1 (empty-env) (diff-cont2 3 (end-cont))))
- = (trampoline (lambda () (apply-cont (diff-cont2 3 (end-cont)) 1)))
- = (trampoline (apply-cont (diff-cont2 3 (end-cont)) 1))
- = (trampoline (lambda () (apply-cont (end-cont) 2)))
- = (trampoline (apply-cont (end-cont) 2))
- = (trampoline (lambda () 2))
- = (trampoline 2)

- Let illustrate how this works

```
(trampoline (value-of-program (value-of/k -(3, 1) (empty-env) (end-cont))))  
• = (trampoline (lambda () (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont)))))  
• = (trampoline (value-of/k 3 (diff1-cont 1 (empty-env) (end-cont))))  
• = (trampoline (lambda () (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3)))  
• = (trampoline (apply-cont (diff1-cont 1 (empty-env) (end-cont)) 3))  
• = (trampoline (lambda () (value-of/k 1 (empty-env) (diff-cont2 3 (end-cont)))))  
• = (trampoline (value-of/k 1 (empty-env) (diff-cont2 3 (end-cont))))  
• = (trampoline (lambda () (apply-cont (diff-cont2 3 (end-cont)) 1)))  
• = (trampoline (apply-cont (diff-cont2 3 (end-cont)) 1))  
• = (trampoline (lambda () (apply-cont (end-cont) 2)))  
• = (trampoline (apply-cont (end-cont) 2))  
• = (trampoline (lambda () 2))  
• = (trampoline 2)  
• = 2
```

- Trampoline input value:
  - A bounce is either:
    1. `expval`
    2. A zero-input function

- Trampoline input value:
  - A bounce is either:
    1. `expval`
    2. A zero-input function
- Specification:
  - If input is an `expval` then return it (computation is done)
  - If input is a zero-input function then evaluate and call the trampoline



- Trampoline input value:

A bounce is either:

1. `expval`
2. A zero-input function

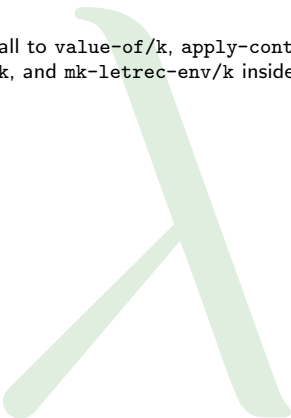
- Specification:

- If input is an `expval` then return it (computation is done)
- If input is a zero-input function then evaluate and call the trampoline

- Trampoline implementation:

```
;; trampoline : bounce → expval  
(define (trampoline a-bounce)  
  (if (expval? a-bounce)  
      a-bounce  
      (trampoline (a-bounce))))
```

- Wrap every call to `value-of/k`, `apply-cont`, `apply-procedure/k`, `eval-rands/k`, and `mk-letrec-env/k` inside a lambda-expression and call `trampoline`



- Wrap every call to `value-of/k`, `apply-cont`, `apply-procedure/k`, `eval-rands/k`, and `mk-letrec-env/k` inside a lambda-expression and call `trampoline`
- `;; value-of-program : Program -> ExpVal`  

```
(define (value-of-program pgm)
  (cases program pgm
    (a-program (exp1)
      (trampoline
        (lambda ()
          (value-of/k exp1 (empty-env) (end-cont)))))))
```

- Wrap every call to `value-of/k`, `apply-cont`, `apply-procedure/k`, `eval-rands/k`, and `mk-letrec-env/k` inside a lambda-expression and call `trampoline`
- ```
;; proc (listof expval) continuation -> bounce
;; Purpose: Apply the given procedure to the given values
(define (apply-procedure/k f vals k)
  (cases proc f
    (procedure (params body envv)
      (let [(saved-env (vector-ref envv 0))]
        (trampoline
         (lambda ()
           (value-of/k body
                        (foldr (lambda (binding acc)
                               (extend-env (car binding)
   (cadr binding)
   acc))
                              saved-env
                              (map (lambda (p v) (list p v))
                                   params
                                   vals))
          k)))))))
```

- Wrap every call to `value-of/k`, `apply-cont`, `apply-procedure/k`, `eval-rands/k`, and `mk-letrec-env/k` inside a lambda-expression and call `trampoline`
- `;; (listof expression) environment continuation --> expval`  
`;; Purpose: Evaluate the given list of exprs and apply the given co`  

```
(define (eval-rands/k rands env k)
  (if (null? rands)
      (trampoline (lambda () (apply-cont k '())))
      (trampoline
        (lambda ()
          (value-of/k
            (car rands) env (eval-rands-cont1 (cdr rands) env k)))))))
```

- Wrap every call to `value-of/k`, `apply-cont`, `apply-procedure/k`, `eval-rands/k`, and `mk-letrec-env/k` inside a lambda-expression and call `trampoline`
- ```
;; (listof symbol) (listof expval) environment continuation --> expval  
;; Purpose: Apply the given cont to the env created using the given  
;; variables, expvals, and environment  
(define (create-let-levn vars vals env k)  
  (if (empty? vars)  
      (trampoline (lambda () (apply-cont k env)))  
      (trampoline  
        (lambda ()  
          (create-let-levn (cdr vars)  
                           (cdr vals)  
                           (extend-env (car vars)  
                                       (car vals) env)  
                                       k))))))
```

- Wrap every call to `value-of/k`, `apply-cont`, `apply-procedure/k`, `eval-rands/k`, and `mk-letrec-env/k` inside a lambda-expression and call `trampoline`
- `;; (listof symbol) (listof (listof symbol)) (listof expression) environment continuation`  
`;; Purpose: Add the proc-vals for the given procedures in the given environment`  
`(define (mk-letrec-env/k names params bodies env k)`  
 `(let* [(temp-proc-vals (map (lambda (p b)`  
 `(proc-val (procedure p b (vector (empty-env))))`  
 `params`  
 `bodies))`  
 `(new-env (foldl (lambda (name proc env)`  
 `(extend-env name`  
 `proc`  
 `env))`  
 `env`  
 `names`  
 `temp-proc-vals))]`  
`(begin`  
 `(for-each (lambda (p)`  
 `(cases proc p`  
 `(procedure (p b ve)`  
 `(vector-set! ve 0 new-env))))`  
 `(map (lambda (p) (expval2proc p))`  
 `temp-proc-vals))`  
`(trampoline (lambda () (apply-cont k new-env))))`

- Wrap every call to value-of/k, apply-cont, apply-procedure/k, eval-rands/k, and mk-letrec-env/k inside a lambda-expression and call trampoline
- ;; expression environment continuation -> bounce

```
(define (value-of/k exp env k)
  (cases expression exp
    (const-exp (num) (trampoline (lambda () (apply-cont k (num-val num))))))

    (true-exp () (trampoline (lambda () (apply-cont k (bool-val #t))))))

    (false-exp () (trampoline (lambda () (apply-cont k (bool-val #f))))))

    (var-exp (var) (trampoline (lambda () (apply-cont k (apply-env env var))))))

    (proc-exp (params body)
      (trampoline
        (lambda ()
          (apply-cont k (proc-val (procedure params body (vector env)))))))

    (zero?-exp (exp1)
      (trampoline (lambda () (value-of/k exp1 env (zero?-cont k))))))

    (diff-exp (exp1 exp2)
      (trampoline
        (lambda () (value-of/k exp1 env (diff-cont1 exp2 env k))))))

    (if-exp (exp1 exp2 exp3)
      (trampoline
        (lambda () (value-of/k exp1 env (if-cont exp2 exp3 env k))))))

    (let-exp (vars exps body)
      (trampoline
        (lambda () (eval-rands/k exps env (let1-cont vars body env k))))))

    (call-exp (rator rands)
      (trampoline
        (lambda () (value-of/k rator env (rator-cont rands env k))))))

    (letrec-exp (names params bodies letrec-body)
      (trampoline
        (lambda ()
          (mk-letrec-env/k names params bodies env (letrec-cont letrec-body k))))))
```



```
• ;; continuation expval --> bounce
;; Purpose: Apply the given cont to the given value and return the final answer
(define (apply-cont k val)
  (cases continuation k
    (end-cont () (trampoline (lambda () val)))
    (zero?-cont (cont)
      (if (zero? (expval2num val))
          (trampoline (lambda () (apply-cont cont (bool-val #t))))
          (trampoline (lambda () (apply-cont cont (bool-val #f))))))
    (let1-cont (vars body env saved-cont)
      (trampoline
        (lambda () (create-let-lev vars val env (let2-cont body saved-cont)))))
    (let2-cont (body saved-cont)
      (trampoline (lambda () (value-of/k body val saved-cont))))
    (if-cont (exp2 exp3 env saved-cont)
      (if (expval2bool val)
          (trampoline (lambda () (value-of/k exp2 env saved-cont)))
          (trampoline (lambda () (value-of/k exp3 env saved-cont)))))
    (letrec-cont (letrec-body saved-cont)
      (trampoline (lambda () (value-of/k letrec-body val saved-cont))))
    (diff-cont1 (exp2 env saved-cont)
      (trampoline (lambda () (value-of/k exp2 env (diff-cont2 val saved-cont)))))
    (diff-cont2 (val1 saved-cont)
      (trampoline
        (lambda () (apply-cont saved-cont (num-val (- (expval2num val1) (expval2num val))))))
      (rator-cont (rands env saved-cont)
        (trampoline
          (lambda () (eval-rands/k rands env (rands-cont val saved-cont)))))
        (rands-cont (rator saved-cont)
          (trampoline
            (lambda () (apply-procedure/k (expval2proc rator) val saved-cont))))
          (eval-rands-cont1 (rands env saved-cont)
            (trampoline
              (lambda () (eval-rands/k rands env (eval-rands-cont2 val saved-cont)))))
            (eval-rands-cont2 (first-rand k)
              (trampoline (lambda () (apply-cont k (cons first-rand val)))))))))
```

- How do we implement a continuation-passing interpreter in procedural languages with assignment?
- Remember that assignment to *shared* variables may be used in place of a binding

## Part V:

### Continuation- Passing Interpreters

Marco T.  
Morazán

Continuation-  
Passing  
Interpreter

A  
Trampoline  
Interpreter

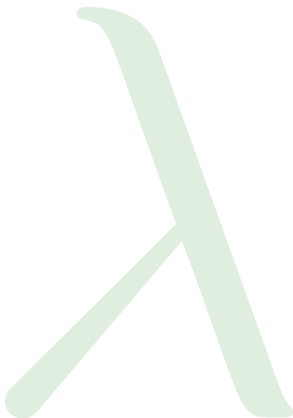
An Imperative  
Interpreter

Exceptions

Threads

- Consider

```
letrec
  even(x) = if zero?(x) then 1 else (odd sub1(x))
  odd(x)   = if zero?(x) then 0 else (even sub1(x))
in (odd 10)
```



## Part V:

### Continuation- Passing Interpreters

Marco T.  
Morazán

### Continuation- Passing Interpreter

A

### Trampoline Interpreter

### An Imperative Interpreter

### Exceptions

### Threads

- Consider

```
letrec
  even(x) = if zero?(x) then 1 else (odd sub1(x))
  odd(x)   = if zero?(x) then 0 else (even sub1(x))
in (odd 10)
```
- Trace

```
(odd 10)
= (even 9)
= (odd 8)
= (even 7)
= (odd 6)
= (even 5)
.
.
.
= (even 1)
= (odd 0)
= 0
```



- Consider

```
letrec
  even(x) = if zero?(x) then 1 else (odd sub1(x))
  odd(x)   = if zero?(x) then 0 else (even sub1(x))
in (odd 10)
```
- Trace

```
(odd 10)
= (even 9)
= (odd 8)
= (even 7)
= (odd 6)
= (even 5)
.
.
.
= (even 1)
= (odd 0)
= 0
```
- Make x a shared variable

```
let x = 10
in letrec
  even() = if zero?(x) then 1 else let d = set x = sub1(x) in (odd)
  odd()  = if zero?(x) then 0 else let d = set x = sub1(x) in (even)
in (odd)
```

- Consider

```
letrec
  even(x) = if zero?(x) then 1 else (odd sub1(x))
  odd(x)   = if zero?(x) then 0 else (even sub1(x))
in (odd 10)
```
- Trace

```
(odd 10)
= (even 9)
= (odd 8)
= (even 7)
= (odd 6)
= (even 5)
.
.
.
= (even 1)
= (odd 0)
= 0
```
- Make x a shared variable

```
let x = 10
in letrec
  even() = if zero?(x) then 1 else let d = set x = sub1(x) in (odd)
  odd()  = if zero?(x) then 0 else let d = set x = sub1(x) in (even)
in (odd)
```
- The trace is the same!

```
(odd )
= (even)
= (odd)
= (even)
= (odd)
= (even)
.
.
.
= (even)
= (odd)
= 0
```

- This means we can rewrite as in assembly using GOTOs and labels

```
x = 10;  
goto odd;  
even:  
  if (x == 0) then return(1) else x = x-1; goto odd;  
odd:  
  if (x == 0) then return(0) else x = x-1; goto even;
```

- This means we can rewrite as in assembly using GOTOs and labels

```
x = 10;  
goto odd;  
even:  
  if (x == 0) then return(1) else x = x-1; goto odd;  
odd:  
  if (x == 0) then return(0) else x = x-1; goto even;
```

- PC trace is the same!

```
(odd)  
= (even)  
= (odd)  
= (even)  
= (odd)  
= (even)  
:  
:  
= (even)  
= (odd)  
= 0
```



- This means we can rewrite as in assembly using GOTOs and labels

```
x = 10;  
goto odd;  
even:  
  if (x == 0) then return(1) else x = x-1; goto odd;  
odd:  
  if (x == 0) then return(0) else x = x-1; goto even;
```

- PC trace is the same!

```
(odd)  
= (even)  
= (odd)  
= (even)  
= (odd)  
= (even)  
:  
:  
= (even)  
= (odd)  
= 0
```

- Why are all these traces the same?

- This means we can rewrite as in assembly using GOTOs and labels

```
x = 10;  
goto odd;  
even:  
  if (x == 0) then return(1) else x = x-1; goto odd;  
odd:  
  if (x == 0) then return(0) else x = x-1; goto even;
```

- PC trace is the same!

```
(odd)  
= (even)  
= (odd)  
= (even)  
= (odd)  
= (even)  
:  
:  
= (even)  
= (odd)  
= 0
```

- Why are all these traces the same?
- Control context does not grow when all function calls are in tail position
- This means a function call is the same as a jump

- Important lessons
- If a set of procs call each other using tail-calls we can rewrite them to use assignment instead of an env
- REMEMBER: Values in one function call do not need to be remembered after another call is made because all calls are tail calls (no delayed operations)

- Important lessons
- If a set of procs call each other using tail-calls we can rewrite them to use assignment instead of an env
- REMEMBER: Values in one function call do not need to be remembered after another call is made because all calls are tail calls (no delayed operations)
- The assignment-based program can be rewritten using GOTOs and labels

- Important lessons
- If a set of procs call each other using tail-calls we can rewrite them to use assignment instead of an env
- REMEMBER: Values in one function call do not need to be remembered after another call is made because all calls are tail calls (no delayed operations)
- The assignment-based program can be rewritten using GOTOs and labels
- We need to identify which procedures need to communicate values
- Use registers to hold those values
- Write an imperative interpreter

- Let's start with our continuation-passing interpreter (not the trampolined interpreter)
- Functions that need to share values via registers:
  - (value-of/k **exp** **env** **cont**)
  - (apply-cont **cont** **val**)
  - (apply-procedure/k **proc1** **val** **cont**)
  - (eval-rands/k **rands** **env** **cont**)
  - (create-let-env **vars** **vals** **env** **cont**)
  - (mk-letrec-env/k **names** **params** **bodies** **env** **cont**)
- How many registers (i.e., state variables) do we need?

- Let's start with our continuation-passing interpreter (not the trampolined interpreter)
- Functions that need to share values via registers:
  - (value-of/k **exp** **env** **cont**)
  - (apply-cont **cont** **val**)
  - (apply-procedure/k **proc1** **val** **cont**)
  - (eval-rands/k **rands** **env** **cont**)
  - (create-let-env **vars** **vals** **env** **cont**)
  - (mk-letrec-env/k **names** **params** **bodies** **env** **cont**)
- How many registers (i.e., state variables) do we need?
- 11 registers given all calls are tail calls
  - exp
  - env
  - cont
  - val
  - proc1
  - rands
  - vars
  - vals
  - letrec-names
  - letrec-params
  - letrec-bodies

- How do 11 registers help us?





- How do 11 registers help us?
- Replace each function with a zero-argument proc
- A function call stores parameter values in the registers and then calls



- How do 11 registers help us?
- Replace each function with a zero-argument proc
- A function call stores parameter values in the registers and then calls
- Three possible scenarios:
  - Register is unchanged then do nothing
  - Make sure field names in cases expression do not shadow a register. If so, rename local vars
  - If a register is used twice in a single call then carefully sequence assignments to have the right values in registers and/or use temporary vars
- This process is called registerization
- From here it is easily translatable into an imperative language (e.g. C)

- ;;; The registers

```
(define exp (void))  
(define env (void))  
(define cont (void))  
(define val (void))  
(define proc1 (void))  
(define rands (void))  
(define vars (void))  
(define vals (void))  
(define letrec-names (void))  
(define letrec-params (void))  
(define letrec-bodies (void))
```

- ```
;; value-of-program : Program -> ExpVal  
(define (value-of-program pgm)  
  (cases program pgm  
    (a-program (exp1)  
      (value-of/k exp1 (empty-env) (end-cont))))))
```
- Becomes  

```
;; value-of-program : Program -> ExpVal  
(define (value-of-program pgm)  
  (cases program pgm  
    (a-program (exp1)  
      (begin  
        (set! exp exp1)  
        (set! env (empty-env))  
        (set! cont (end-cont))  
        (value-of/k))))))
```

- `;; expression environment continuation -> expval`

```
(define (value-of/k)  
  (cases expression exp  
  
    (const-exp (num)  
      (apply-cont k (num-val num))))
```

- Becomes

```
;; expression environment continuation -> expval  
(define (value-of/k)  
  (cases expression exp  
  
    (const-exp (num)  
      (begin  
        (set! val (num-val num))  
        (apply-cont))))
```

- `;; expression environment continuation -> expval`

```
(define (value-of/k)
  (cases expression exp
    (true-exp ()
      (apply-cont k (bool-val #t)))
```

- Becomes

```
;; expression environment continuation -> expval
(define (value-of/k)
  (cases expression exp
    (true-exp ()
      (begin
        (set! val (bool-val #t))
        (apply-cont)))
```

- `;; expression environment continuation -> expval`  
`(define (value-of/k)`  
  `(cases expression exp`  
    `(false-exp ()`  
      `(apply-cont k (bool-val #f)))`  
    `(true-exp ()`  
      `(apply-cont k (bool-val #t)))`  
    `(lambda-exp (env) (lambda-val env))`  
    `(let-exp (env) (lambda-val env))`  
    `(letrec-exp (env) (lambda-val env))`  
    `(call-exp (env) (lambda-val env))`  
    `(error-exp () (error-val #f)))`  
  `)`
- Becomes  
  `(false-exp ()`  
    `(begin`  
      `(set! val (bool-val #f))`  
      `(apply-cont)))`

- ```
;; expression environment continuation -> expval  
(define (value-of/k)  
  (cases expression exp  
  
    (var-exp (var)  
              (apply-cont k (apply-env env var)))
```
- Becomes

```
;; expression environment continuation -> expval  
(define (value-of/k)  
  (cases expression exp  
  
    (var-exp (var)  
              (begin  
                (set! val (apply-env env var))  
                (apply-cont)))
```



- ```
;; expression environment continuation -> expval  
(define (value-of/k)  
  (cases expression exp  
  
    (zero?-exp (exp1)  
                (value-of/k exp1 env (zero?-cont k)))
```
- Becomes  

```
;; expression environment continuation -> expval  
(define (value-of/k)  
  (cases expression exp  
  
    (zero?-exp (exp1)  
                (begin  
                  (set! exp exp1)  
                  (set! cont (zero?-cont cont))  
                  (value-of/k)))
```

- ```
;; expression environment continuation -> expval  
(define (value-of/k)  
  (cases expression exp  
  
    (diff-exp (exp1 exp2)  
              (value-of/k exp1 env (diff-cont1 exp2 env k)))
```
- Becomes

```
;; expression environment continuation -> expval  
(define (value-of/k)  
  (cases expression exp  
  
    (diff-exp (exp1 exp2)  
              (begin  
                (set! exp exp1)  
                (set! cont (diff-cont1 exp2 env cont))  
                (value-of/k)))
```

- ```
;; expression environment continuation -> expval  
(define (value-of/k)  
  (cases expression exp  
  
    (if-exp (exp1 exp2 exp3)  
      (value-of/k exp1 env (if-cont exp2 exp3 env k)))
```
- Becomes  

```
;; expression environment continuation -> expval  
(define (value-of/k)  
  (cases expression exp  
  
    (if-exp (exp1 exp2 exp3)  
      (begin  
        (set! exp exp1)  
        (set! cont (if-cont exp2 exp3 env cont))  
        (value-of/k)))
```

- ```
;; expression environment continuation -> expval  
(define (value-of/k)  
  (cases expression exp  
  
    (let-exp (vars exps body)  
              (eval-rands/k exps env (let1-cont vars body env k)))
```
- Becomes  

```
;; expression environment continuation -> expval  
(define (value-of/k)  
  (cases expression exp  
  
    (let-exp (vars exps body)  
              (begin  
                (set! rands exps)  
                (set! cont (let1-cont vars body env cont))  
                (eval-rands/k)))
```

- ```
;; expression environment continuation -> expval  
(define (value-of/k)  
  (cases expression exp  
  
    (call-exp (rator rands)  
              (value-of/k rator env (rator-cont rands env k)))
```
- Becomes

```
;; expression environment continuation -> expval  
(define (value-of/k)  
  (cases expression exp  
  
    (call-exp (rator rands)  
              (begin  
                (set! exp rator)  
                (set! cont (rator-cont rands env cont))  
                (value-of/k)))
```

- `;; expression environment continuation -> expval`  
`(define (value-of/k)`  
  `(cases expression exp`  
  
    `(letrec-exp (names params bodies letrec-body)`  
      `(mk-letrec-env/k`  
        `names params bodies env (letrec-cont letrec-body k))`  
      `names params bodies env (letrec-cont letrec-body k))`  
    `)`  
`)`
- Becomes  
`;; expression environment continuation -> expval`  
`(define (value-of/k)`  
  `(cases expression exp`  
  
    `(letrec-exp (names params bodies letrec-body)`  
      `(begin`  
        `(set! letrec-names names)`  
        `(set! letrec-params params)`  
        `(set! letrec-bodies bodies)`  
        `(set! cont (letrec-cont letrec-body cont))`  
        `(mk-letrec-env/k)))`  
      `names params bodies env (letrec-cont letrec-body k))`  
    `)`  
`)`

- ```
(define (mk-letrec-env/k)
  (let* [...]
    (begin
      (for-each (lambda (p)
                  (cases proc p
                    (procedure (p b ve)
                              (vector-set! ve 0 new-env))))
                (map (lambda (p) (expval2proc p))
                     temp-proc-vals))
      (apply-cont k new-env))))
```
- Becomes

```
(define (mk-letrec-env/k)
  (let* [...]
    (begin
      (for-each (lambda (p)
                  (cases proc p
                    (procedure (p b ve)
                              (vector-set! ve 0 new-env))))
                (map (lambda (p) (expval2proc p))
                     temp-proc-vals))
      (begin
        (set! val new-env)
        (apply-cont))))))
```

- ```
(define (create-let-letenv)
  (if (empty? vars)
      (apply-cont k env)
      (create-let-letenv (cdr vars)
                          (cdr vals)
                          (extend-env (car vars) (car vals) env)
                          k)))
```
- Becomes

```
(define (create-let-letenv)
  (if (empty? vars)
      (begin
        (set! val env)
        (apply-cont))
      (begin
        ;; beware of dependencies between registers
        (set! env (extend-env (car vars) (car vals) env))
        (set! vars (cdr vars))
        (set! vals (cdr vals))
        (create-let-letenv))))
```



- ```
(define (eval-rands/k)
  (if (null? rands)
      (apply-cont k '())
      (value-of/k (car rands)
                  env
                  (eval-rands-cont1 (cdr rands) env k))))
```
- Becomes

```
(define (eval-rands/k)
  (if (null? rands)
      (begin
        (set! val '())
        (apply-cont))
      (begin
        (set! exp (car rands))
        (set! cont (eval-rands-cont1 (cdr rands) env cont))
        (value-of/k)))))
```

- ```
(define (apply-cont)
  (cases continuation cont
    (end-cont () val)

    (zero?-cont (saved-cont)
      (if (zero? (expval2num val))
          (apply-cont saved-cont (bool-val #t))
          (apply-cont cont (bool-val #f)))))
```
- Becomes

```
(define (apply-cont)
  (cases continuation cont
    (end-cont () val) ;; no function call, no change

    (zero?-cont (saved-cont)
      (if (zero? (expval2num val))
          (begin
            (set! val (bool-val #t))
            (set! cont saved-cont)
            (apply-cont))
          (begin
            (set! val (bool-val #f))
            (set! cont saved-cont)
            (apply-cont)))))
```

- (define (apply-cont)  
 (cases continuation cont

```
    (let1-cont (the-vars body saved-env saved-cont)
      (create-let-lenv vars
        val
        saved-env
        (let2-cont body saved-cont))))
```

- Becomes

```
(define (apply-cont)
  (cases continuation cont
```

```
    (let1-cont (the-vars body saved-env saved-cont)
      (begin
        (set! vars the-vars)
        (set! vals val) ;; val is the list of evaluated RHSs
        (set! env saved-env)
        (set! cont (let2-cont body saved-cont))
        (create-let-lenv))))
```

- ```
(define (apply-cont)
  (cases continuation cont

    (let2-cont (body saved-cont)
      (value-of/k body val saved-cont))
```
- Becomes

```
(define (apply-cont)
  (cases continuation cont

    (let2-cont (body saved-cont)
      (begin
        (set! exp body)
        (set! env val) ;; val is an env
        (set! cont saved-cont)
        (value-of/k))))
```

- ```
(define (apply-cont)
  (cases continuation cont

    (if-cont (exp2 exp3 saved-env saved-cont)
      (if (expval2bool val)
        (value-of/k exp2 saved-env saved-cont)
        (value-of/k exp3 env saved-cont)))

    Becomes

    (define (apply-cont)
      (cases continuation cont

        (if-cont (exp2 exp3 saved-env saved-cont)
          (if (expval2bool val)
            (begin
              (set! exp exp2)
              (set! env saved-env)
              (set! cont saved-cont)
              (value-of/k))
            (begin
              (set! exp exp3)
              (set! cont saved-cont)
              (value-of/k))))))
```

- ```
(define (apply-cont)
  (cases continuation cont

    (letrec-cont (letrec-body saved-cont)
      (value-of/k letrec-body val saved-cont))
```
- Becomes

```
(define (apply-cont)
  (cases continuation cont

    (letrec-cont (letrec-body saved-cont)
      (begin
        (set! exp letrec-body)
        (set! env val) ;; val is an env
        (set! cont saved-cont)
        (value-of/k))))
```

- ```
(define (apply-cont)
  (cases continuation cont

    (diff-cont1 (exp2 saved-env saved-cont)
      (value-of/k exp2 saved-env (diff-cont2 val saved-cont)))

    )
```
- Becomes

```
(define (apply-cont)
  (cases continuation cont

    (diff-cont1 (exp2 saved-env saved-cont)
      (begin
        (set! exp exp2)
        (set! env saved-env)
        (set! cont (diff-cont2 val saved-cont))
        (value-of/k)))

    )
```

- ```
(define (apply-cont)
  (cases continuation cont

    (diff-cont2 (val1 saved-cont)
      (apply-cont saved-cont (num-val (- (expval2num val1)
                                          (expval2num val))))))
```
- Becomes

```
(define (apply-cont)
  (cases continuation cont

    (diff-cont2 (val1 saved-cont)
      (begin
        (set! cont saved-cont)
        (set! val (num-val (- (expval2num val1)
                              (expval2num val))))
        (apply-cont))))
```



- ```
(define (apply-cont)
  (cases continuation cont
    (rator-cont (saved-rands saved-env saved-cont)
      (eval-rands/k
        saved-rands saved-env (rands-cont val saved-cont))))
```
- Becomes

```
(define (apply-cont)
  (cases continuation cont
    (rator-cont (saved-rands saved-env saved-cont)
      (begin
        (set! rands saved-rands)
        (set! env saved-env)
        (set! cont (rands-cont val saved-cont))
        (eval-rands/k))))
```

- ```
(define (apply-cont)
  (cases continuation cont

    (rands-cont (rator saved-cont)
      (apply-procedure/k (expval2proc rator) val saved-cont))
```
- Becomes

```
(define (apply-cont)
  (cases continuation cont

    (rands-cont (rator saved-cont)
      (begin
        (set! proc1 (expval2proc rator))
        (set! vals val) ;; val is the list of evaluated args
        (set! cont saved-cont)
        (apply-procedure/k)))
```



- ```
(define (apply-cont)
  (cases continuation cont

    (eval-rands-cont2 (first-rand saved-cont)
      (apply-cont saved-cont (cons first-rand val)))
```
- Becomes

```
(define (apply-cont)
  (cases continuation cont

    (eval-rands-cont2 (first-rand saved-cont)
      (begin
        (set! cont saved-cont)
        (set! val (cons first-rand val))
        (apply-cont))))
```

- Add mult-exp and add-exp to the imperative interpreter
- 5.29

- Remember exceptions (or throwing errors)?

X

- Remember exceptions (or throwing errors)?
- Manage ordinary control flow
- Alter the control context
- Continuations may be used for these

- Remember exceptions (or throwing errors)?
- Manage ordinary control flow
- Alter the control context
- Continuations may be used for these
- Two new type of expressions in our language



- Remember exceptions (or throwing errors)?
- Manage ordinary control flow
- Alter the control context
- Continuations may be used for these
- Two new type of expressions in our language
- `expression`  $\rightarrow$  `try expression catch (identifier) expression`
- Evaluate first expression and if it returns normally then its value is the value of the `try-expression` and the handler expression is ignored

- Remember exceptions (or throwing errors)?
- Manage ordinary control flow
- Alter the control context
- Continuations may be used for these
- Two new type of expressions in our language
- `expression`  $\rightarrow$  `try expression catch (identifier) expression`
- Evaluate first expression and if it returns normally then its value is the value of the `try`-expression and the handler expression is ignored
- `expression`  $\rightarrow$  `raise expression`
- Evaluate the expression and send that value to the most recent exception handler

- Syntax

```
(define the-grammar
  '((program (expression) a-program)
    (expression (number) const-exp)
    ...
    (expression ("try" expression "catch" "(" identifier ")") expression)
    (expression ("raise" expression) raise-exp)
  ))
```

- ```
(eval
  "let f = proc (n)      ;; function to return n if not 0
    if zero?(n)
    then raise -1
    else n
  in try (f 0) catch (i) -(i, 1)") ;; handler subs 1 from its input
```

- `expression`  $\rightarrow$  `try expression catch (identifier) expression`
- Need a continuation for what to do after evaluating the first expression  
(define-datatype continuation cont?

```
  :  
(try-cont  
  (var symbol?)  
  (handler-exp expression?) ;; use if abnormal evaluation end  
  (env environment?)  
  (cont continuation?))
```

- `expression`  $\rightarrow$  `try expression catch (identifier) expression`
- Need a continuation for what to do after evaluating the first expression  
(`define-datatype continuation cont?`)

```
      :  
      (try-cont  
        (var symbol?)  
        (handler-exp expression?) ;; use if abnormal evaluation end  
        (env environment?)  
        (cont continuation?))
```

- `expression`  $\rightarrow$  `raise expression`
- Need a continuation for what to do after evaluating the expression
- (`define-datatype continuation cont?`)

```
      :  
      (try-cont  
        (var symbol?)  
        (handler-exp expression?) ;; use if abnormal evaluation end  
        (env environment?)  
        (cont cont?))
```

```
(raise1-cont (saved-cont cont?))  
;; save current cont to find handler (saved in a try-cont)
```

- ```
(define (value-of/k exp env k)
  (cases expression exp
    :
    (try-exp (exp1 var handler)
      (value-of/k exp1 env (try-cont var handler env k)))
```

- ```
(define (value-of/k exp env k)
  (cases expression exp
    :
    (try-exp (exp1 var handler)
      (value-of/k exp1 env (try-cont var handler env k)))
```
- ```
(raise-exp (exp1)
  (value-of/k exp1 env (raise1-cont k)))
```



- ```
(define (apply-cont k val)
  (cases continuation k
    :
    ;; no exception, proceed normally with saved continuation
    (try-cont (var handler-exp saved-env saved-cont)
      (apply-cont saved-cont val))

    ;; exception raised, find handler and apply
    (raise1-cont (saved-cont)
      (apply-handler val saved-cont))
```

- `;; apply-handler: expval cont → Final Answer`  
`(define (apply-handler val cont)`  
  `(cases continuation cont`  
    `(try-cont (var handler-exp saved-env saved-cont)`  
      `(value-of/k`  
        `handler-exp ;; evaluate handler`  
        `(extend-env var val saved-env) ;; binding for exception value`  
        `saved-cont))`  
    `(end-cont () (report-uncaught-exception)) ;; no handler`  
    `(else (apply-handler val saved-cont)))) ;; keep searching`

- 5.37, 5.38



- One may want to have multiple computations at the same time in the same address space
- Text editor: Spell checker, Back-up, Grammar check, etc.
- Same address space and same process: threads

- One may want to have multiple computations at the same time in the same address space
- Text editor: Spell checker, Back-up, Grammar check, etc.
- Same address space and same process: threads
- Interpreter will simulate the execution of multiple threads
- Each thread is a computation in progress
- Threads communicate through shared memory using assignment

- Big picture (much from your OS course)
- Pool of threads
- Each thread: running, runnable, blocked



- Big picture (much from your OS course)
- Pool of threads
- Each thread: running, runnable, blocked
- In our system, only one thread running at a time
- Ready Queue: contains the runnable threads
- Scheduler chooses which thread to run
- Preemptive scheduling (quantum or time slice)
- Start with **IMPLICIT-REFS Language**

- Big picture (much from your OS course)
- Pool of threads
- Each thread: running, runnable, blocked
- In our system, only one thread running at a time
- Ready Queue: contains the runnable threads
- Scheduler chooses which thread to run
- Preemptive scheduling (quantum or time slice)
- Start with **IMPLICIT-REFS Language**
- spawn: creates a new thread
  - takes one arg that should evaluate to a proc
  - When a thread is run an argument is passed to this proc
  - Does not run immediately. . . .put in the ready queue
  - spawn is executed for effect; don't care return value



- Non-cooperating threads

```
letrec
  noisy (l) = if null?(l)
              then 0
              else begin
                  print(car(l));
                  (noisy cdr(l))
              end
in begin
    spawn(proc (d) (noisy [1, 2, 3, 4, 5]));
    spawn(proc (d) (noisy [6, 7, 8, 9, 10]));
    print(100);
    33
end
```

- Non-cooperating threads

```
letrec
  noisy (l) = if null?(l)
              then 0
              else begin
                    print(car(l));
                    (noisy cdr(l))
                  end
in begin
  spawn(proc (d) (noisy [1, 2, 3, 4, 5]));
  spawn(proc (d) (noisy [6, 7, 8, 9, 10]));
  print(100);
  33
end
```

- A trace

```
100  main
1    first thread
2
3
6    second thread
7
8
4    first thread
5
9    second thread
10
returns 33 (when threads are done)
```

- Cooperating threads examples

```
let buffer = 0 one element buffer shared

in let producer = proc (n) input: a buffer element n
    letrec wait(k) =
        if zero?(k)
        then set buffer = n
        else begin print(-(k,-200));
                    (wait -(k,1)) end
    in (wait 5) execute wait loop 5 times
    ignores input, loops while buffer is 0 printing the timer
in let consumer = proc (d)
    letrec busywait (k) =
        if zero?(buffer)
        then begin print(-(k,-100));
                    (busywait -(k,-1)) end
        else buffer
        in (busywait 0)
    in begin spawn(proc (d) (producer 44));
        print(300);
        (consumer 86) end
```

- Cooperating threads examples

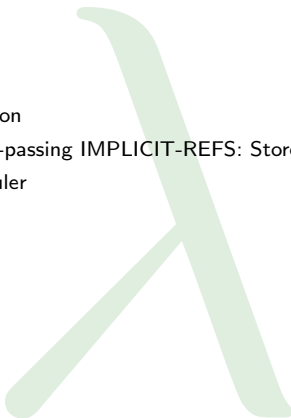
```
let buffer = 0 one element buffer shared

in let producer = proc (n) input: a buffer element n
    letrec wait(k) =
        if zero?(k)
        then set buffer = n
        else begin print(-(k,-200));
                    (wait -(k,1)) end
    in (wait 5) execute wait loop 5 times
    ignores input, loops while buffer is 0 printing the timer
in let consumer = proc (d)
    letrec busywait (k) =
        if zero?(buffer)
        then begin print(-(k,-100));
                    (busywait -(k,-1)) end
        else buffer
        in (busywait 0)
    in begin spawn(proc (d) (producer 44));
        print(300);
        (consumer 86) end
```

- Trace

```
300
100   consumer runs for a while
101
205   producer runs for a while
204
203
102   consumer runs for a while
103
202   producer runs for a while
201
104   consumer runs for a while
105
producer sets buffer to 44 and stops
consumer returns 44
```

- Implementation
- Continuation-passing IMPLICIT-REFS: Store + continuations
- Add a scheduler



- Scheduler
- Internal State
  - Ready queue
  - Final answer – value of main thread when done
  - Max time slice – max steps a thread can run
  - Time remaining – steps left for the running thread

- Scheduler
- Internal State
  - Ready queue
  - Final answer – value of main thread when done
  - Max time slice – max steps a thread can run
  - Time remaining – steps left for the running thread
- Scheduler Interface
  - `init-scheduler: int → void`
  - `place-on-ready-queue!: thread → void`
  - `run-next-thread: () → FinalAnswer`, runs next thread or if none returns the final answer
  - `set-final-answer!: expval → void`
  - `time-expired?: () → Bool`
  - `decrement-timer!: () → void`

- Queue Interface

```
(define (empty-queue) '())
```

```
(define null? null?)
```

```
(define (enqueue q val) (append q (list val)))
```

```
(define (dequeue q f) (f (car q) (cdr q)))
```

- f updates the state of the scheduler and runs the first thread



- Queue Interface

```
(define (empty-queue) '())
```

```
(define null? null?)
```

```
(define (enqueue q val) (append q (list val)))
```

```
(define (dequeue q f) (f (car q) (cdr q)))
```

- f updates the state of the scheduler and runs the first thread

- State variables (registers)

```
(define the-ready-queue 'uninitialized)
```

```
(define the-final-answer 'uninitialized)
```

```
(define the-max-time-slice 'uninitialized)
```

```
(define the-time-remaining 'uninitialized)
```

- Scheduler Implementation

```
;; natnum>0 → void
;; Purpose: Initialize the scheduler
(define (initialize-scheduler! ticks)
  (begin
    (set! the-ready-queue (empty-queue))
    (set! the-final-answer 'uninitialized)
    (set! the-max-time-slice ticks)
    (set! the-time-remaining the-max-time-slice)))
```

- Scheduler Implementation

```
;; natnum>0 → void
;; Purpose: Initialize the scheduler
(define (initialize-scheduler! ticks)
  (begin
    (set! the-ready-queue (empty-queue))
    (set! the-final-answer 'uninitialized)
    (set! the-max-time-slice ticks)
    (set! the-time-remaining the-max-time-slice)))

• ;; thread → (void)
  ;; Purpose: Place given thread in the ready queue
  (define (place-on-ready-queue! th)
    (set! the-ready-queue (enqueue the-ready-queue th)))
```

- Scheduler Implementation

```
;; natnum>0 → void
;; Purpose: Initialize the scheduler
(define (initialize-scheduler! ticks)
  (begin
    (set! the-ready-queue (empty-queue))
    (set! the-final-answer 'uninitialized)
    (set! the-max-time-slice ticks)
    (set! the-time-remaining the-max-time-slice)))

• ;; thread → (void)
  ;; Purpose: Place given thread in the ready queue
  (define (place-on-ready-queue! th)
    (set! the-ready-queue (enqueue the-ready-queue th)))

• ;; expval → (void)
  ;; Purpose: Set the final answer register
  (define (set-final-answer! val) (set! the-final-answer val))
```

- Scheduler Implementation

```
;; natnum>0 → void
;; Purpose: Initialize the scheduler
(define (initialize-scheduler! ticks)
  (begin
    (set! the-ready-queue (empty-queue))
    (set! the-final-answer 'uninitialized)
    (set! the-max-time-slice ticks)
    (set! the-time-remaining the-max-time-slice)))

• ;; thread → (void)
  ;; Purpose: Place given thread in the ready queue
  (define (place-on-ready-queue! th)
    (set! the-ready-queue (enqueue the-ready-queue th)))

• ;; expval → (void)
  ;; Purpose: Set the final answer register
  (define (set-final-answer! val) (set! the-final-answer val))

• ;; → Bool
  ;; Purpose: Determine if time slice has ended
  (define (time-expired?) (zero? the-time-remaining))
```

- Scheduler Implementation

```
;; natnum>0 → void
;; Purpose: Initialize the scheduler
(define (initialize-scheduler! ticks)
  (begin
    (set! the-ready-queue (empty-queue))
    (set! the-final-answer 'uninitialized)
    (set! the-max-time-slice ticks)
    (set! the-time-remaining the-max-time-slice)))

• ;; thread → (void)
  ;; Purpose: Place given thread in the ready queue
  (define (place-on-ready-queue! th)
    (set! the-ready-queue (enqueue the-ready-queue th)))

• ;; expval → (void)
  ;; Purpose: Set the final answer register
  (define (set-final-answer! val) (set! the-final-answer val))

• ;; → Bool
  ;; Purpose: Determine if time slice has ended
  (define (time-expired?) (zero? the-time-remaining))

• ;; → (void)
  ;; Purpose: Decrement the time slice
  (define (decrement-timer!)
    (set! the-time-remaining (- the-time-remaining 1)))
```

- Scheduler Implementation

- ```
;; natnum>0 → void
;; Purpose: Initialize the scheduler
(define (initialize-scheduler! ticks)
  (begin
    (set! the-ready-queue (empty-queue))
    (set! the-final-answer 'uninitialized)
    (set! the-max-time-slice ticks)
    (set! the-time-remaining the-max-time-slice)))
```
- ```
;; thread → (void)
;; Purpose: Place given thread in the ready queue
(define (place-on-ready-queue! th)
  (set! the-ready-queue (enqueue the-ready-queue th)))
```
  - ```
;; expval → (void)
;; Purpose: Set the final answer register
(define (set-final-answer! val) (set! the-final-answer val))
```
  - ```
;; → Bool
;; Purpose: Determine if time slice has ended
(define (time-expired?) (zero? the-time-remaining))
```
  - ```
;; → (void)
;; Purpose: Decrement the time slice
(define (decrement-timer!)
  (set! the-time-remaining (- the-time-remaining 1)))
```
  - ```
;; → expval
;; Purpose: Run the next thread in the ready queue
(define (run-next-thread)
  (if (empty-queue? the-ready-queue)
      the-final-answer
      (dequeue the-ready-queue
        (lambda (first-ready-thread other-ready-threads)
          (set! the-ready-queue other-ready-threads)
          (set! the-time-remaining the-max-time-slice)
          (first-ready-thread))))))
```

- Threads
- `thread = () → expval`
- a procedure with no args that returns an `expval`



- Interpreter
- IMPLICIT-REFS refactored to be a continuation-passing interpreter (as we did for LETREC)
- `(spawn-exp (exp) (value-of/k exp env (spawn-cont k)))`

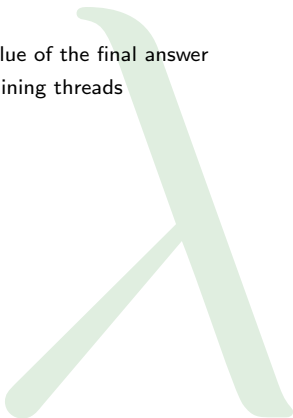
- Interpreter
- IMPLICIT-REFS refactored to be a continuation-passing interpreter (as we did for LETREC)
- (spawn-exp (exp) (value-of/k exp env (spawn-cont k)))
- apply-cont

```
(spawn-cont (saved-cont)
  (let ((proc1 (expval2proc val)))
    (begin
      (place-on-ready-queue!
        (lambda ()
          (apply-procedure/k
            proc1
            (list (num-val 28)) ;; apply to dummy val
            (end-subthread-cont))))
      (apply-cont saved-cont (num-val 73)))))) ;; return dummy val
```

- Interpreter
- What continuation should each thread be run in?



- Interpreter
- What continuation should each thread be run in?
- Main thread
- record the value of the final answer
- run any remaining threads



- Interpreter
- What continuation should each thread be run in?
- Main thread
- record the value of the final answer
- run any remaining threads
- apply-cont

```
(end-main-thread-cont ()  
  (begin  
    (set-final-answer! val)  
    (run-next-thread)))
```

- Interpreter
- What continuation should each thread be run in?
- Main thread
  - record the value of the final answer
  - run any remaining threads
  - apply-cont

```
(end-main-thread-cont ()  
  (begin  
    (set-final-answer! val)  
    (run-next-thread)))
```
- Other subthreads
  - ignore its value
  - runs remaining threads

- Interpreter
- What continuation should each thread be run in?
- Main thread
- record the value of the final answer
- run any remaining threads
- apply-cont

```
(end-main-thread-cont ()  
  (begin  
    (set-final-answer! val)  
    (run-next-thread)))
```

- Other subthreads
- ignore its value
- runs remaining threads
- apply-cont

```
(end-subthread-cont ()  
  (run-next-thread))
```

- To evaluate a program:
- initialize the store
- initialize the scheduler
- evaluate expression in the `end-main-thread-cont`





- To evaluate a program:
- initialize the store
- initialize the scheduler
- evaluate expression in the end-main-thread-cont
- `;; value-of-program : natnum program → expval`  
`(define (value-of-program timeslice pgm)`  
  `(initialize-store!)`  
  `(initialize-scheduler! timeslice)`  
  `(cases program pgm`  
    `(a-program (exp1)`  
      `(begin`  
        `(value-of/k`  
          `exp1`  
          `(empty-env)`  
          `(end-main-thread-cont))`  
      `the-final-answer))))`

- To evaluate a program:
- initialize the store
- initialize the scheduler
- evaluate expression in the end-main-thread-cont
- ```
;; value-of-program : natnum program → expval
(define (value-of-program timeslice pgm)
  (initialize-store!)
  (initialize-scheduler! timeslice)
  (cases program pgm
    (a-program (exp1)
      (begin
        (value-of/k
          exp1
          (empty-env)
          (end-main-thread-cont))
        the-final-answer))))
```
- The wrapper function (eval)  

```
(define TIMESLICE 5)

;; string → ExpVal
;; Purpose: Evaluate the given extended LC-program
(define (eval string)
  (value-of-program TIMESLICE (parse string)))
```

- Applying a continuation
- If time has expired then place thread on ready queue interrupting the continuation application and run the next thread



- Applying a continuation
- If time has expired then place thread on ready queue interrupting the continuation application and run the next thread
- If time has not expired then decrement the timer and apply the given continuation to the given value

- Applying a continuation
- If time has expired then place thread on ready queue interrupting the continuation application and run the next thread
- If time has not expired then decrement the timer and apply the given continuation to the given value
- ```
(define (apply-cont k val)
  (if (time-expired?)
      (begin
        (place-on-ready-queue! (lambda () (apply-cont k val)))
        (run-next-thread))
      (begin
        (decrement-timer!)
        (cases continuation k
          :
          :
          :))))
```

- ```
(eval "let x = 0
      in let mut = mutex()
        in let incr_x = proc (id) proc (dummy)
          begin set x = -(x, -1);
                print(x)
          end
        in begin spawn((incr_x 100));
                spawn((incr_x 200));
                spawn((incr_x 300))
        end" )
```
- What is the value of x when the program ends?

- ```
(eval "let x = 0
      in let mut = mutex()
         in let incr_x = proc (id) proc (dummy)
            begin set x = -(x, -1);
                  print(x)
            end
         in begin spawn((incr_x 100));
                  spawn((incr_x 200));
                  spawn((incr_x 300))
            end")
```
- What is the value of x when the program ends?
- ```
set x = -(x, -1);
LDR1 x
LDR2 -1
SUB
SET X R3
```

- ```
(eval "let x = 0
      in let mut = mutex()
      in let incr_x = proc (id) proc (dummy)
        begin set x = -(x, -1);
        print(x)
      end
      in begin spawn((incr_x 100));
        spawn((incr_x 200));
        spawn((incr_x 300))
      end")
```
- What is the value of x when the program ends?
- set x = -(x, -1);

```
LDR1 x
LDR2 -1
SUB
SET X R3
```
- Trace

```
T1 runs and sets x to 1
T2 runs and sets x to 2
T3 runs and sets x to 3
x is 3 after all threads run
```
- Trace

```
T1 LDR1 x Loads 0 as the value of x
LDR2 -1
T1 is interrupted
```



- ```

      (eval "let x = 0
              in let mut = mutex()
                  in let incr_x = proc (id) proc (dummy)
                      begin set x = -(x, -1);
                          print(x)
                      end
                  in begin spawn((incr_x 100));
                      spawn((incr_x 200));
                      spawn((incr_x 300))
                  end")
    
```
- What is the value of x when the program ends?
- ```

      set x = -(x, -1);
      LDR1 x
      LDR2 -1
      SUB
      SET X R3
    
```
- Trace
  - T1 runs and sets x to 1
  - T2 runs and sets x to 2
  - T3 runs and sets x to 3
  - x is 3 after all threads run**
- Trace
  - T1 LDR1 x **Loads 0 as the value of x**
  - LDR2 -1
  - T1 is interrupted
- T2 LDR1 x **Loads 0 as the value of x**
  - LDR2 -1
  - SUB
  - SET X R2 **mutates x to 1**

- ```

      (eval "let x = 0
              in let mut = mutex()
                  in let incr_x = proc (id) proc (dummy)
                      begin set x = -(x, -1);
                          print(x)
                      end
                  in begin spawn((incr_x 100));
                      spawn((incr_x 200));
                      spawn((incr_x 300))
                  end" )
    
```
- What is the value of x when the program ends?
- ```

      set x = -(x, -1);
      LDR1 x
      LDR2 -1
      SUB
      SET X R3
    
```
- Trace
  - T1 runs and sets x to 1
  - T2 runs and sets x to 2
  - T3 runs and sets x to 3
  - x is 3 after all threads run**
- Trace
  - ```

          T1 LDR1 x  Loads 0 as the value of x
             LDR2 -1
             T1 is interrupted
        
```
  - ```

          T2 LDR1 x  Loads 0 as the value of x
             LDR2 -1
             SUB
             SET X R2  mutates x to 1
        
```
  - ```

          T2 LDR1 x  Loads 1 as the value of x
             LDR2 -1
             SUB
             SET X R3  mutates x to 2
        
```



- How do we avoid such interferences between processes?



- How do we avoid such interferences between processes?
- Synchronization mechanism is needed

- How do we avoid such interferences between processes?
- Synchronization mechanism is needed
- Binary semaphore or mutex: used to control access to shared variables and avoid busy waiting

- How do we avoid such interferences between processes?
- Synchronization mechanism is needed
- Binary semaphore or mutex: used to control access to shared variables and avoid busy waiting
- Mutex
  - is open or closed
  - has a queue of threads waiting for the mutex to become open

- Mutex interface
- mutex
  - no args
  - creates an open mutex





- Mutex interface
- mutex
  - no args
  - creates an open mutex
- wait
  - one argument
  - used to indicate access to a mutex
  - open mutex → mutex becomes closed and thread runs
  - closed mutex → thread put in the mutex's queue
  - returns void

- Mutex interface
- mutex
  - no args
  - creates an open mutex
- wait
  - one argument
  - used to indicate access to a mutex
  - open mutex → mutex becomes closed and thread runs
  - closed mutex → thread put in the mutex's queue
  - returns void
- signal
  - single mutex argument
  - used to release a mutex
  - mutex closed and empty queue → mutex open and thread continues
  - mutex closed and non-empty queue → one thread put in ready queue, queue remains closed, and thread continues executing
  - returns no value

- Mutex interface
- mutex
  - no args
  - creates an open mutex
- wait
  - one argument
  - used to indicate access to a mutex
  - open mutex → mutex becomes closed and thread runs
  - closed mutex → thread put in the mutex's queue
  - returns void
- signal
  - single mutex argument
  - used to release a mutex
  - mutex closed and empty queue → mutex open and thread continues
  - mutex closed and non-empty queue → one thread put in ready queue, queue remains closed, and thread continues executing
  - returns no value
- Mutex guarantees
  - only one thread has access to shared vars between wait and signal calls
  - region where shared vars are accessed is called a *critical region/section*

- (eval  
 "let x = 0  
 in let mut = mutex()  
 in let incr\_x =  
 proc (id)  
 proc (dummy)  
 begin  
 wait(mut);  
 set x = -(x, -1); critical section  
 print(x);  
 signal(mut)  
 end  
 in begin  
 spawn((incr\_x 100));  
 spawn((incr\_x 200));  
 spawn((incr\_x 300))  
 end")

- Implementation

```
(define-datatype mutex mutex?  
  (a-mutex  
    (ref-to-closed? reference?)      ;; ref to bool closed or open  
    (ref-to-wait-queue reference?))) ;; ref to (listof thread)
```



- Implementation

```
(define-datatype mutex mutex?  
  (a-mutex  
    (ref-to-closed? reference?)      ;; ref to bool closed or open  
    (ref-to-wait-queue reference?))) ;; ref to (listof thread)
```

- Mutexes are expressed values

```
(define-datatype expval expval?  
  (num-val  
    (value number?))  
  (bool-val  
    (boolean boolean?))  
  (proc-val  
    (proc proc?))  
  (mutex-val  
    (mutex mutex?)))  
  
(define expval2mutex  
  (lambda (v)  
    (cases expval v  
      (mutex-val (m) m)  
      (else (expval-extractor-error 'mutex v)))))
```

- Implementation

```
(define-datatype mutex mutex?  
  (a-mutex  
    (ref-to-closed? reference?)      ;; ref to bool closed or open  
    (ref-to-wait-queue reference?))) ;; ref to (listof thread)
```

- Mutexes are expressed values

```
(define-datatype expval expval?  
  (num-val  
    (value number?))  
  (bool-val  
    (boolean boolean?))  
  (proc-val  
    (proc proc?))  
  (mutex-val  
    (mutex mutex?)))  
  
(define expval2mutex  
  (lambda (v)  
    (cases expval v  
      (mutex-val (m) m)  
      (else (expval-extractor-error 'mutex v)))))
```

- In value-of/k

```
(mutex-exp () (apply-cont k (mutex-val (new-mutex))))
```

- (define (new-mutex) (a-mutex (newref #f) (newref '())))

- wait
- In value-of/k

```
(wait-exp (exp) (value-of/k exp env (wait-cont k)))
```



- wait
- In value-of/k  

```
(wait-exp (exp) (value-of/k exp env (wait-cont k)))
```
- In apply-cont  

```
(wait-cont (saved-cont)  
  (wait-for-mutex  
    (expval2mutex val)  
    (lambda () (apply-cont saved-cont (num-val 52))))))
```

- wait
- In value-of/k  

```
(wait-exp (exp) (value-of/k exp env (wait-cont k)))
```
- In apply-cont  

```
(wait-cont (saved-cont)  
  (wait-for-mutex  
    (expval2mutex val)  
    (lambda () (apply-cont saved-cont (num-val 52))))))
```
- signal
- In value-of/k  

```
(signal-exp (exp) (value-of/k exp env (signal-cont k)))
```

- wait  
`(wait-exp (exp) (value-of/k exp env (wait-cont k)))`
- In value-of/k  
`(wait-exp (exp) (value-of/k exp env (wait-cont k)))`
- In apply-cont  
`(wait-cont (saved-cont)  
 (wait-for-mutex  
 (expval2mutex val)  
 (lambda () (apply-cont saved-cont (num-val 52))))))`
- signal
- In value-of/k  
`(signal-exp (exp) (value-of/k exp env (signal-cont k)))`
- In apply-cont  
`(signal-cont (saved-cont)  
 (signal-mutex (expval2mutex val)  
 (lambda ()  
 (apply-cont saved-cont (num-val 53)))))`

- ```
;; mutex thread → expval
;; Purpose: waits for mutex to be open.
(define (wait-for-mutex m th)
  (cases mutex m
    (a-mutex (ref-to-closed? ref-to-wait-queue)
      (cond [(deref ref-to-closed?)
              (setref! ref-to-wait-queue
                        (enqueue (deref ref-to-wait-queue) th))
              (run-next-thread)]
            [else (begin
                      (setref! ref-to-closed? #t)
                      (th))])))
```

- ```
;; mutex thread → expval
;; Purpose: To signal given mutex and run a waiting thread if any
(define (signal-mutex m th)
  (cases mutex m
    (a-mutex (ref-to-closed? ref-to-wait-queue)
      (let [(closed? (deref ref-to-closed?))
            (wait-queue (deref ref-to-wait-queue))]
        (if closed?
          (begin
            (if (empty? wait-queue)
              (setref! ref-to-closed? #f)
              (dequeue
               wait-queue
               (lambda (first-waiting-th other-waiting-ths)
                 (begin
                  (place-on-ready-queue! first-waiting-th)
                  (setref! ref-to-wait-queue other-waiting-ths))))))
            (th)))
          ;;error in book: Page 190, Fig 5.22
          (th))))))
```

- 5.45, 5.47, 5.58 (outline the algorithm!)

