

Part III: Expressions

Marco T. Morazán

Seton Hall University

Outline

- 1 A Language Based on an Extended Lambda Calculus
- 2 Evaluation Wrapper Functions and Tests
- 3 Expressed and Denoted Values
- 4 Environment Datatype
- 5 Language Specification
- 6 Implementation
- 7 Variable Names Elimination

A Language
Based on an
Extended
Lambda
Calculus

Evaluation
Wrapper
Functions and
Tests

Expressed and
Denoted
Values

Environment
Datatype

Language
Specification

Implementation

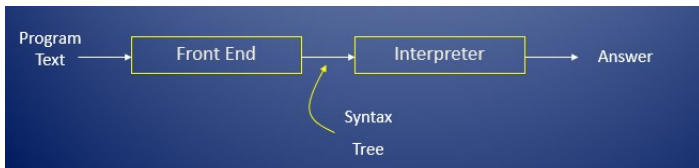
Variable
Names
Elimination

Automatic Parsing

- We will now study the semantics (or meaning) of some fundamental programming languages features
- Our primary tools will be interpreters

Automatic Parsing

- We will now study the semantics (or meaning) of some fundamental programming languages features
- Our primary tools will be interpreters



Automatic Parsing

- Adapted and Extended Lambda Calculus

```
<exp> ::= <number>
       ::= true
       ::= false
       ::= <id>
       ::= -(<exp>, <exp>)
       ::= zero?(<exp>)
       ::= if <exp> then <exp> else <exp>
       ::= let {<id> = <exp>}* in <exp>
       ::= proc(<id>*) <exp>
       ::= (<exp> <exp>*)
       ::= letrec {identifier (<id>*) = <exp>}* in <exp>
```

Automatic Parsing

- Adapted and Extended Lambda Calculus

```
<exp> ::= <number>
      ::= true
      ::= false
      ::= <id>
      ::= -(<exp>, <exp>)
      ::= zero?(<exp>)
      ::= if <exp> then <exp> else <exp>
      ::= let {<id> = <exp>}* in <exp>
      ::= proc(<id>*) <exp>
      ::= (<exp> <exp>*)
      ::= letrec {identifier (<id>*) = <exp>}* in <exp>
```

- We will use a parser generator system: `slgen`
- Read: Appendix B

Automatic Parsing

- Boilerplate Code

```
#lang eopl
```

```
(require rackunit "../eopl-extras.rkt")
```

```
;;;;;;;;;;;;;; grammatical specification ;;;;;;;;;;;;;;
```

```
(define the-lexical-spec
```

```
  '((whitespace (whitespace) skip)
```

```
    (comment ("% (arbn (not /newline))) skip)
```

```
    (identifier
```

```
      (letter (arbn (or letter digit "_" "-" "?"))) symbol)
```

```
    (number (digit (arbn digit)) number)
```

```
    (number ("-" digit (arbn digit)) number)))
```

Automatic Parsing

- The Grammar (not boilerplate)
`'((program (expression) a-program)`

Automatic Parsing

- The Grammar (not boilerplate)
 - '((program (expression) a-program)
 - (expression (number) const-exp)
(expression ("true") true-exp)
(expression ("false") false-exp)
(expression (identifier) var-exp)

Automatic Parsing

- The Grammar (not boilerplate)
 - '((program (expression) a-program)
 - (expression (number) const-exp)
(expression ("true") true-exp)
(expression ("false") false-exp)
(expression (identifier) var-exp)
 - (expression("-" "(" expression "," expression ")")diff-exp)

Automatic Parsing

- The Grammar (not boilerplate)
 - '((program (expression) a-program)
 - (expression (number) const-exp)
(expression ("true") true-exp)
(expression ("false") false-exp)
(expression (identifier) var-exp)
 - (expression("-" "(" expression "," expression ")")diff-exp)
 - (expression ("zero?" "(" expression ")") zero?-exp)

Automatic Parsing

- The Grammar (not boilerplate)
 - '((program (expression) a-program)
 - (expression (number) const-exp)
(expression ("true") true-exp)
(expression ("false") false-exp)
(expression (identifier) var-exp)
 - (expression("-" "(" expression "," expression ")")diff-exp)
 - (expression ("zero?" "(" expression ")") zero?-exp)
 - (expression
("if" expression "then" expression "else" expression) if-exp)

Automatic Parsing

- The Grammar (not boilerplate)
 - '((program (expression) a-program)
 - (expression (number) const-exp)
(expression ("true") true-exp)
(expression ("false") false-exp)
(expression (identifier) var-exp)
 - (expression("-" "(" expression "," expression ")")diff-exp)
 - (expression ("zero?" "(" expression ")") zero?-exp)
 - (expression
("if" expression "then" expression "else" expression) if-exp)
 - (expression
("let"(arbn identifier)="expression)"in" expression) let-exp)

Automatic Parsing

- The Grammar (not boilerplate)
 - '((program (expression) a-program)
 - (expression (number) const-exp)
 - (expression ("true") true-exp)
 - (expression ("false") false-exp)
 - (expression (identifier) var-exp)
 - (expression("-" "(" expression "," expression ")")diff-exp)
 - (expression ("zero?" "(" expression ")") zero?-exp)
 - (expression
("if" expression "then" expression "else" expression) if-exp)
 - (expression
("let"(arbno identifier)="expression)"in" expression) let-exp)
 - (expression
("proc" "(" (arbno identifier) ")" expression) proc-exp)

Automatic Parsing

- The Grammar (not boilerplate)
 - '((program (expression) a-program)
 - (expression (number) const-exp)
 - (expression ("true") true-exp)
 - (expression ("false") false-exp)
 - (expression (identifier) var-exp)
 - (expression("-" "(" expression "," expression ")")diff-exp)
 - (expression ("zero?" "(" expression ")") zero?-exp)
 - (expression
("if" expression "then" expression "else" expression) if-exp)
 - (expression
("let"(arbno identifier)="expression)"in" expression) let-exp)
 - (expression
("proc" "(" (arbno identifier) ")" expression) proc-exp)
 - (expression ("(" expression (arbno expression) ")") call-exp)))

Automatic Parsing

- The Grammar (not boilerplate)
 - '((program (expression) a-program)
 - (expression (number) const-exp)
 - (expression ("true") true-exp)
 - (expression ("false") false-exp)
 - (expression (identifier) var-exp)
 - (expression("-" "(" expression "," expression ")")diff-exp)
 - (expression ("zero?" "(" expression ")") zero?-exp)
 - (expression
("if" expression "then" expression "else" expression) if-exp)
 - (expression
("let"(arbn identifier)="expression)"in" expression) let-exp)
 - (expression
("proc" "(" (arbn identifier) ")" expression) proc-exp)
 - (expression ("(" expression (arbn expression) ")") call-exp)))
 - (expression
("letrec"(arbn identifier)(" (arbn identifier))""="expression)
"in" expression) letrec-exp)

Automatic Parsing

- Boilerplate

```
(sllgen:make-define-datatypes the-lexical-spec the-grammar)
```

```
(define show-the-datatypes  
  (lambda ()  
    (sllgen:list-define-datatypes the-lexical-spec the-grammar)))
```

```
(define scan&parse  
  (sllgen:make-string-parser the-lexical-spec the-grammar))
```

```
(define just-scan  
  (sllgen:make-string-scanner the-lexical-spec the-grammar))
```

Automatic Parsing

- ```
> (show-the-datatypes)
((define-datatype program program? (a-program (a-program21 expression?)))
 (define-datatype
 expression
 expression?
 (const-exp (const-exp22 number?))
 (true-exp)
 (false-exp)
 (var-exp (var-exp23 symbol?))
 (diff-exp (diff-exp24 expression?) (diff-exp25 expression?))
 (zero?-exp (zero?-exp26 expression?))
 (if-exp (if-exp27 expression?) (if-exp28 expression?) (if-exp29 expression?))
 (let-exp
 (let-exp30 (list-of symbol?))
 (let-exp31 (list-of expression?))
 (let-exp32 expression?))
 (letrec-exp
 (letrec-exp33 (list-of symbol?))
 (letrec-exp34 (list-of (list-of symbol?)))
 (letrec-exp35 (list-of expression?))
 (letrec-exp36 expression?))
 (proc-exp (proc-exp37 (list-of symbol?)) (proc-exp38 expression?))
 (call-exp (call-exp39 expression?) (call-exp40 (list-of expression?)))))
```

# Automatic Parsing

- > (show-the-datatypes)  
((define-datatype program program? (a-program (a-program21 expression?)))  
 (define-datatype  
 expression  
 expression?  
 (const-exp (const-exp22 number?))  
 (true-exp)  
 (false-exp)  
 (var-exp (var-exp23 symbol?))  
 (diff-exp (diff-exp24 expression?) (diff-exp25 expression?))  
 (zero?-exp (zero?-exp26 expression?))  
 (if-exp (if-exp27 expression?) (if-exp28 expression?) (if-exp29 expression?))  
 (let-exp  
 (let-exp30 (list-of symbol?))  
 (let-exp31 (list-of expression?))  
 (let-exp32 expression?))  
 (letrec-exp  
 (letrec-exp33 (list-of symbol?))  
 (letrec-exp34 (list-of (list-of symbol?)))  
 (letrec-exp35 (list-of expression?))  
 (letrec-exp36 expression?))  
 (proc-exp (proc-exp37 (list-of symbol?)) (proc-exp38 expression?))  
 (call-exp (call-exp39 expression?) (call-exp40 (list-of expression?)))))
- > (scan&parse "if 0 then 1 else 2")  
#(struct:a-program  
 #(struct:if-exp #(struct:const-exp 0) #(struct:const-exp 1) #(struct:const-exp 2)))

# Evaluation Wrapper Functions and Tests

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

- `;; string → a-program`  
`;; Purpose: Parse the given extended LC-program`  
`(define (parse p) (scan&parse p))`

# Evaluation Wrapper Functions and Tests

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

- `;; string → a-program`  
`;; Purpose: Parse the given extended LC-program`  
`(define (parse p) (scan&parse p))`
- `;; string → expval`  
`;; Purpose: Evaluate the given extended LC-program`  
`(define (eval string)`  
`(value-of-program (parse string)))`

# Evaluation Wrapper Functions and Tests

- ```
(check-equal? (eval "if zero?(1) then 1 else 2") (num-val 2))
(check-equal? (eval "-(15, 10)") (num-val 5))
(check-equal?
  (eval "let x = 10 in if zero?(-(x, x)) then x else 2")
  (num-val 10))
(check-equal? (eval "let decr = proc (a) -(a, 1) in (decr 30)")
  (num-val 29))
(check-equal? (eval "( proc (g) (g 30) proc (y) -(y, 1))")
  (num-val 29))
(check-equal? (eval "let x = 200
  in let f = proc (z) -(z, x)
  in let x = 100
  in let g = proc (z) -(z, x)
  in -((f 1), (g 1))")
  (num-val -100))
(check-equal? (eval "let sum = proc (x) proc (y) -(x, -(0, y)) in ((sum 3) 4)")
  (num-val 7))
(check-equal? (eval "let sum = proc (x) proc (y) -(x, -(0, y))
  in letrec sigma (n) = if zero?(n)
  then 0
  else ((sum n) (sigma -(n, 1)))
  in (sigma 5)")
  (num-val 15))
(check-equal? (eval "letrec even(n) = if zero?(n)
  then zero?(n)
  else if zero?(-(n, 1))
  then zero?(n)
  else (even -(n, 2))
  in (even 501)")
  (bool-val #f)))
```

Expressed and Denoted Values

- An important part of the specification of any programming language is the set of values the language manipulates

A Language
Based on an
Extended
Lambda
Calculus

Evaluation
Wrapper
Functions and
Tests

**Expressed and
Denoted
Values**

Environment
Datatype

Language
Specification

Implementation

Variable
Names
Elimination

Expressed and Denoted Values

- An important part of the specification of any programming language is the set of values the language manipulates
- Each language has at least two sets:
 - Expressed values Possible values of expressions
 - Denoted values Values bound to variables

Expressed and Denoted Values

- An important part of the specification of any programming language is the set of values the language manipulates
- Each language has at least two sets:
 - Expressed values Possible values of expressions
 - Denoted values Values bound to variables
- In our extended λ -calculus
 - Expressed values: Int, Bool, and Proc
 - Denoted values: Int, Bool, Proc

Expressed and Denoted Values

- An important part of the specification of any programming language is the set of values the language manipulates

- Each language has at least two sets:

Expressed values Possible values of expressions

Denoted values Values bound to variables

- In our extended λ -calculus

- Expressed values: Int, Bool, and Proc
- Denoted values: Int, Bool, Proc

- We need an interface for expressed values

Constructors

- num-val: $\text{int} \rightarrow \text{expval}$
- bool-val: $\text{Boolean} \rightarrow \text{expval}$
- proc-val: $(\text{listof symbol}) \text{ expression env} \rightarrow \text{expval}$

Observers

- expval- \rightarrow int : $\text{expval} \rightarrow \text{int}$
- expval- \rightarrow bool : $\text{expval} \rightarrow \text{boolean}$
- expval- \rightarrow proc : $\text{expval} \rightarrow \text{proc}$

Expressed and Denoted Values

- Expressed values

```
(define-datatype expval expval?  
  (num-val (value number?))  
  (bool-val (boolean boolean?))  
  (proc-val (proc proc?)))
```

Expressed and Denoted Values

- Expressed values

```
(define-datatype expval expval?  
  (num-val (value number?))  
  (bool-val (boolean boolean?))  
  (proc-val (proc proc?)))
```

- Extractors

```
;; expval → Int throws error  
;; Purpose: Extract number from given expval  
(define (expval2num v)  
  (cases expval v  
    (num-val (num) num)  
    (else (expval-extractor-error 'num-val v))))
```

Expressed and Denoted Values

- Expressed values

```
(define-datatype expval expval?  
  (num-val (value number?))  
  (bool-val (boolean boolean?))  
  (proc-val (proc proc?)))
```

- Extractors

```
;; expval → Int throws error  
;; Purpose: Extract number from given expval  
(define (expval2num v)  
  (cases expval v  
    (num-val (num) num)  
    (else (expval-extractor-error 'num-val v))))
```

- ;; expval → Bool throws error
;; Purpose: Extract Boolean from given expval
(define (expval2bool v)
 (cases expval v
 (bool-val (bool) bool)
 (else (expval-extractor-error 'bool-val v))))

Expressed and Denoted Values

- Expressed values

```
(define-datatype expval expval?  
  (num-val (value number?))  
  (bool-val (boolean boolean?))  
  (proc-val (proc proc?)))
```

- Extractors

```
;; expval → Int throws error  
;; Purpose: Extract number from given expval  
(define (expval2num v)  
  (cases expval v  
    (num-val (num) num)  
    (else (expval-extractor-error 'num-val v))))
```

- ;; expval → Bool throws error
;; Purpose: Extract Boolean from given expval
(define (expval2bool v)
 (cases expval v
 (bool-val (bool) bool)
 (else (expval-extractor-error 'bool-val v))))

- ;; expval → proc throws error
;; Purpose: Extract proc from given expval
(define (expval2proc v)
 (cases expval v
 (proc-val (proc) proc)
 (else (expval-extractor-error 'proc-val v))))

Expressed and Denoted Values

- Expressed values

```
(define-datatype expval expval?  
  (num-val (value number?))  
  (bool-val (boolean boolean?))  
  (proc-val (proc proc?)))
```

- Extractors

```
;; expval → Int throws error  
;; Purpose: Extract number from given expval  
(define (expval2num v)  
  (cases expval v  
    (num-val (num) num)  
    (else (expval-extractor-error 'num-val v))))
```

- ;; expval → Bool throws error
;; Purpose: Extract Boolean from given expval
(define (expval2bool v)
 (cases expval v
 (bool-val (bool) bool)
 (else (expval-extractor-error 'bool-val v))))

- ;; expval → proc throws error
;; Purpose: Extract proc from given expval
(define (expval2proc v)
 (cases expval v
 (proc-val (proc) proc)
 (else (expval-extractor-error 'proc-val v))))

- ;; symbol expval → throws error
;; Purpose: Throw expval extraction error
(define (expval-extractor-error variant value)
 (eopl:error 'expval-extractors "Looking for a ~s, given ~s"
 variant value))

Environment

- Having vars → need for an environment
- Discussed earlier in this course

Environment

- Having vars \rightarrow need for an environment
- Discussed earlier in this course
- Notation
 - $[x = 3][y = 7][u = 5]\rho$
 - $[x = 3, y = 7, u = 5]\rho$
 - $(\text{extend-env } 'x\ 3\ (\text{extend-env } 'y\ 7\ (\text{extend-env } 'u\ 5\ \rho)))$

Environment

- Having vars \rightarrow need for an environment
- Discussed earlier in this course
- Notation
 - $[x = 3][y = 7][u = 5]\rho$
 - $[x = 3, y = 7, u = 5]\rho$
 - $(\text{extend-env } 'x\ 3\ (\text{extend-env } 'y\ 7\ (\text{extend-env } 'u\ 5\ \rho)))$
- ```
(define-datatype environment environment?
 (empty-env)
 (extend-env
 (bvar symbol?)
 (bval expval?)
 (saved-env environment?)))
```

# Environment

- Having vars  $\rightarrow$  need for an environment
- Discussed earlier in this course
- Notation
  - $[x = 3][y = 7][u = 5]\rho$
  - $[x = 3, y = 7, u = 5]\rho$
  - $(\text{extend-env } 'x\ 3\ (\text{extend-env } 'y\ 7\ (\text{extend-env } 'u\ 5\ \rho)))$
- ```
(define-datatype environment environment?  
  (empty-env)  
  (extend-env  
    (bvar symbol?)  
    (bval expval?)  
    (saved-env environment?)))
```
- ```
(define (apply-env env search-sym)
 (cases environment env
 (empty-env ()
 (eopl:error 'apply-env "No binding for ~s" search-sym))
 (extend-env (var val saved-env)
 (if (eqv? search-sym var)
 val
 (apply-env saved-env search-sym)))))
```

# Language Specification

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

- Specifying the behavior of programs
  - Constructor** a-program:  $\text{expression} \rightarrow \text{program}$
  - Observer** (value-of-program  $e$ ) = (value-of  $e$   $\rho_{init}$ )
- Specifying the behavior of expressions
  - Constructors** const-exp, true-exp, false-exp, diff-exp, etc.
  - Observer** value-of:  $\text{expression env} \rightarrow \text{expval}$

# Language Specification

- **value-of:**

# Language Specification

- **value-of:**
- $(\text{value-of } (\text{const-exp } n) \rho) = (\text{num-val } n)$

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

# Language Specification

- **value-of:**
- $(\text{value-of } (\text{const-exp } n) \rho) = (\text{num-val } n)$
- $(\text{value-of } (\text{true-exp } n) \rho) = (\text{bool-val } \#t)$
- $(\text{value-of } (\text{false-exp } n) \rho) = (\text{bool-val } \#f)$

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

# Language Specification

- **value-of:**
- $(\text{value-of } (\text{const-exp } n) \rho) = (\text{num-val } n)$
- $(\text{value-of } (\text{true-exp } n) \rho) = (\text{bool-val } \#t)$
- $(\text{value-of } (\text{false-exp } n) \rho) = (\text{bool-val } \#f)$
- $(\text{value-of } (\text{var-exp } n) \rho) = (\text{apply-env } \rho \ x)$

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination



# Language Specification

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

- **value-of:**
- $(\text{value-of } (\text{const-exp } n) \rho) = (\text{num-val } n)$
- $(\text{value-of } (\text{true-exp } n) \rho) = (\text{bool-val } \#t)$
- $(\text{value-of } (\text{false-exp } n) \rho) = (\text{bool-val } \#f)$
- $(\text{value-of } (\text{var-exp } n) \rho) = (\text{apply-env } \rho \ x)$
- $$\frac{v1=(\text{expval2num } (\text{value-of } e1 \ \rho)) \ \wedge \ v2=(\text{expval2num } (\text{value-of } e2 \ \rho))}{(\text{diff-exp } e1 \ e2) = (\text{num-val } (- \ v1 \ v2))}$$

# Language Specification

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

- **value-of:**
- $(\text{value-of } (\text{const-exp } n) \rho) = (\text{num-val } n)$
- $(\text{value-of } (\text{true-exp } n) \rho) = (\text{bool-val } \#t)$
- $(\text{value-of } (\text{false-exp } n) \rho) = (\text{bool-val } \#f)$
- $(\text{value-of } (\text{var-exp } n) \rho) = (\text{apply-env } \rho \ x)$
- $$\frac{v1=(\text{expval2num } (\text{value-of } e1 \ \rho)) \ \wedge \ v2=(\text{expval2num } (\text{value-of } e2 \ \rho))}{(\text{diff-exp } e1 \ e2) = (\text{num-val } (- \ v1 \ v2))}$$
- $$\frac{v=(\text{expval2num } (\text{value-of } e) \rho)}{(\text{value-of } (\text{zero?-exp } n) \ \rho) = \begin{cases} (\text{bool-val } \#t), & \text{if } v = 0 \\ (\text{bool-val } \#f), & \text{if } v \neq 0 \end{cases}}$$

# Language Specification

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

- **value-of:**
- $(\text{value-of } (\text{const-exp } n) \rho) = (\text{num-val } n)$
- $(\text{value-of } (\text{true-exp } n) \rho) = (\text{bool-val } \#t)$
- $(\text{value-of } (\text{false-exp } n) \rho) = (\text{bool-val } \#f)$
- $(\text{value-of } (\text{var-exp } n) \rho) = (\text{apply-env } \rho \ x)$
- $$\frac{v1=(\text{expval2num } (\text{value-of } e1 \ \rho)) \ \wedge \ v2=(\text{expval2num } (\text{value-of } e2 \ \rho))}{(\text{diff-exp } e1 \ e2) = (\text{num-val } (- \ v1 \ v2))}$$
- $$\frac{v=(\text{expval2num } (\text{value-of } e) \ \rho)}{(\text{value-of } (\text{zero?-exp } n) \ \rho) = \begin{cases} (\text{bool-val } \#t), & \text{if } v = 0 \\ (\text{bool-val } \#f), & \text{if } v \neq 0 \end{cases}}$$
- $$\frac{cval=(\text{expval2bool } (\text{value-of } c \ \rho))}{(\text{value-of } (\text{if-exp } c \ t \ e) \ \rho) = \begin{cases} (\text{value-of } t \ \rho), & \text{if } cval = \#t \\ (\text{value-of } e \ \rho), & \text{if } cval = \#f \end{cases}}$$

# Language Specification

- To evaluate a let-expression the environment must be extended with the bindings for the local variables before evaluation the body

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

# Language Specification

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

- To evaluate a let-expression the environment must be extended with the bindings for the local variables before evaluation the body

- $$\frac{(v1 \dots vn) = (\text{map } (\text{lambda } (e) \text{ (value-of } e \ \rho)) (e1 \dots en))}{(\text{value-of } (\text{let-exp } (s1 \dots sn) (e1 \dots en) \text{ body}) \rho) = (\text{value-of } \textit{body} [s1=v1 \dots sn=vn] \rho)}$$

# Language Specification

- For a language to be useful, it must allow the creation of new procedures
- - `expval = int + boolean + proc`
  - `denval = int + boolean + proc`
- `proc` is a set of values representing procedures

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

# Language Specification

- For a language to be useful, it must allow the creation of new procedures
- - `expval = int + boolean + proc`
  - `denvval = int + boolean + proc`
- `proc` is a set of values representing procedures
- What should this evaluate to?

```
let decr = proc (a) -(a, 1)
in (decr 30)
```

# Language Specification

- For a language to be useful, it must allow the creation of new procedures

- - `expval = int + boolean + proc`
  - `denval = int + boolean + proc`

- `proc` is a set of values representing procedures

- What should this evaluate to?

```
let decr = proc (a) -(a, 1)
in (decr 30)
```

- What should this evaluate to?

```
(proc (g) (g 30) proc (y) -(y, 1))
```



# Language Specification

- For a language to be useful, it must allow the creation of new procedures

- $\text{expval} = \text{int} + \text{boolean} + \text{proc}$   
 $\text{denval} = \text{int} + \text{boolean} + \text{proc}$

- `proc` is a set of values representing procedures

- What should this evaluate to?

```
let decr = proc (a) -(a, 1)
in (decr 30)
```

- What should this evaluate to?

```
(proc (g) (g 30) proc (y) -(y, 1))
```

- What should this evaluate to?

```
let x = 200
in let f = proc (z) -(z, x)
in let x = 100
in let g = proc (z) -(z, x)
in -((f 1), (g 1))
```

# Language Specification

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

- What should this evaluate to?

```
 let x = 200
 in let f = proc (z) -(z, x)
 in let x = 100 Same expression behaves differently
 in let g = proc (z) -(z, x)
 in -((f 1), (g 1))
```

- Variables must obey the lexical binding rule
- The value of a proc-exp depends on the environment

# Language Specification

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

- What should this evaluate to?

```
 let x = 200
 in let f = proc (z) -(z, x)
 in let x = 100 Same expression behaves differently
 in let g = proc (z) -(z, x)
 in -((f 1), (g 1))
```

- Variables must obey the lexical binding rule
- The value of a proc-exp depends on the environment
- $(\text{value-of } (\text{proc-exp } (p1 \dots pn) b) \rho) = (\text{proc-val } (\text{procedure } (p1 \dots pn) b \rho))$

# Language Specification

- What needs to happen to evaluate a call-exp?

```
let x = 200
in let f = proc (z) -(z, x)
 in let x = 100
 in let g = proc (z) -(z, x)
 in -((f 1), (g 1))
```

# Language Specification

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

- What needs to happen to evaluate a call-exp?

```
let x = 200
in let f = proc (z) -(z, x)
 in let x = 100
 in let g = proc (z) -(z, x)
 in -(f 1), (g 1))
```

- - 1 Evaluate f
  - 2 Evaluate 1
  - 3 Apply the proc to its argument(s)

# Language Specification

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

- What needs to happen to evaluate a call-exp?

```
let x = 200
in let f = proc (z) -(z, x)
 in let x = 100
 in let g = proc (z) -(z, x)
 in -(f 1), (g 1))
```

- ① Evaluate f
- ② Evaluate 1
- ③ Apply the proc to its argument(s)
- $$\frac{p = (\text{expval} \rightarrow \text{proc}(\text{value-of } e0 \ \rho)) \wedge \text{args} = (\text{map } (\text{lambda } (e) (\text{value-of } e \ \text{env})) (e1 \dots en))}{(\text{value-of } (\text{call-exp } e0 \ (e1 \dots en)) \ \rho) = (\text{apply-procedure } p \ \text{args})}$$

# Language Specification

- Recursion

```
letrec fact(n) = if zero?(n)
 then 1
 else *(n, (fact {(n, 1)}))
in (fact 5)
```

- (value-of letrec-exp(p-names params p-bodys letrec-body)  $\rho$ ) = ???

# Language Specification

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

- Recursion

```
letrec fact(n) = if zero?(n)
 then 1
 else *(n, (fact {(n, 1)}))
in (fact 5)
```

- $(\text{value-of letrec-exp}(p\text{-names } params \text{ } p\text{-bodys letrec-body}) \rho) = ???$
- $(\text{value-of letrec-body } ???)$
- What env is needed to evaluate the body?



# Language Specification

- Recursion

```
letrec fact(n) = if zero?(n)
 then 1
 else *(n, (fact {(n, 1)}))
in (fact 5)
```

- $(\text{value-of letrec-exp}(p\text{-names } params \text{ } p\text{-bodys letrec-body}) \rho) = ???$
- $(\text{value-of letrec-body } ???)$
- What env is needed to evaluate the body?
- $\rho?$

# Language Specification

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

- Recursion

```
letrec fact(n) = if zero?(n)
 then 1
 else *(n, (fact {(n, 1)}))
in (fact 5)
```

- $(\text{value-of letrec-exp}(p\text{-names } params \text{ p-bodys letrec-body}) \rho) = ???$
- $(\text{value-of letrec-body } ???)$
- What env is needed to evaluate the body?
- $\rho?$
- $(\text{value-of (fact 5)} \rho)$   
= error fact is not in the env

# Language Specification

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

- Recursion

```
letrec fact(n) = if zero?(n)
 then 1
 else *(n, (fact {(n, 1)}))
in (fact 5)
```

- $(\text{value-of letrec-exp}(\text{p-names params p-bodys letrec-body}) \rho) = ???$
- $(\text{value-of letrec-body } ???)$
- What env is needed to evaluate the body?
- $\rho?$
- $(\text{value-of (fact 5)} \rho)$   
= **error fact is not in the env**
- Body of letrec-exp cannot be evaluated using  $\rho$

# Language Specification

- Recursion

```
letrec fact(n) = if zero?(n) then 1 else *(n, (fact {(n, 1)}))
in (fact 5)
```
- (value-of (letrec-exp p-names params p-bodys letrec-body)  $\rho$ ) = ???

# Language Specification

- Recursion  
letrec fact(n) = if zero?(n) then 1 else \*(n, (fact {(n, 1)}))  
in (fact 5)
- (value-of (letrec-exp p-names params p-bodys letrec-body)  $\rho$ ) = ???
- = (value-of letrec-body ???)
- What env is needed to evaluate the body?

# Language Specification

- Recursion  
letrec fact(n) = if zero?(n) then 1 else \*(n, (fact (n, 1)))  
in (fact 5)
- (value-of (letrec-exp p-names params p-bodys letrec-body)  $\rho$ ) = ???
- = (value-of letrec-body ???)
- What env is needed to evaluate the body?
- We need an env that contains a binding for all the local functions defined

# Language Specification

- Recursion  
letrec fact(n) = if zero?(n) then 1 else \*(n, (fact {(n, 1)}))  
in (fact 5)
- (value-of (letrec-exp p-names params p-bodys letrec-body)  $\rho$ ) = ???
- = (value-of letrec-body ???)
- What env is needed to evaluate the body?
- We need an env that contains a binding for all the local functions defined
- Create the procedure first?  
(value-of  
  (fact 5)  
  [fact=(procedure '(n) (if zero?(n) then 1 else \*(n, (fact {(n, 1)}))) ?]  $\rho$ )

# Language Specification

- Recursion

```
letrec fact(n) = if zero?(n) then 1 else *(n, (fact {(n, 1)}))
in (fact 5)
```
- $(\text{value-of } (\text{letrec-exp } p\text{-names } params \text{ } p\text{-bodies } \text{letrec-body}) \rho) = ???$
- $= (\text{value-of } \text{letrec-body } ???)$
- What env is needed to evaluate the body?
- We need an env that contains a binding for all the local functions defined
- Create the procedure first?

```
(value-of
 (fact 5)
 [fact=(procedure '(n) (if zero?(n) then 1 else *(n, (fact {(n, 1)})) ?)] ρ)
```
- $= (\text{value-of}$ 

```
 (fact 5)
 [fact=(procedure
 '(n)
 (if zero?(n) then 1 else *(n, (fact {(n, 1)}))
 ρ)]
 ρ)
```



# Language Specification

- Recursion
  - letrec fact(n) = if zero?(n) then 1 else \*(n, (fact {(n, 1)}))
  - in (fact 5)
- (value-of (letrec-exp p-names params p-bodys letrec-body)  $\rho$ ) = ???
- = (value-of letrec-body ???)
- What env is needed to evaluate the body?
- We need an env that contains a binding for all the local functions defined
- Create the procedure first?
  - (value-of
    - (fact 5)
    - [fact=(procedure '(n) (if zero?(n) then 1 else \*(n, (fact {(n, 1)})) ?)]  $\rho$ )
- = (value-of
  - (fact 5)
  - [fact=(procedure
    - '(n)
    - (if zero?(n) then 1 else \*(n, (fact {(n, 1)})) $\rho$ )] $\rho$ )
- = (value-of
  - \* (n, (fact {(n, 1)}))
  - [n=5]
  - [fact=(procedure
    - '(n)
    - (if zero?(n) then 1 else \*(n, (fact {(n, 1)})) $\rho$ )] $\rho$ )

# Language Specification

- Recursion
  - letrec fact(n) = if zero?(n) then 1 else \*(n, (fact {(n, 1)}))
  - in (fact 5)
- (value-of (letrec-exp p-names params p-bodys letrec-body)  $\rho$ ) = ???
- = (value-of letrec-body ???)
- What env is needed to evaluate the body?
- We need an env that contains a binding for all the local functions defined
- Create the procedure first?
  - (value-of
    - (fact 5)
      - [fact=(procedure '(n) (if zero?(n) then 1 else \*(n, (fact {(n, 1)}))) ?)]  $\rho$ )
  - = (value-of
    - (fact 5)
      - [fact=(procedure
        - '(n)
          - (if zero?(n) then 1 else \*(n, (fact {(n, 1)})))
        - $\rho$ ]
    - $\rho$ )
  - = (value-of
    - \* (n, (fact {(n, 1)}))
      - [n=5]
        - [fact=(procedure
          - '(n)
            - (if zero?(n) then 1 else \*(n, (fact {(n, 1)})))
          - $\rho$ ]
      - $\rho$ )
  - = (value-of \*(n, (fact {(n, 1)})) [n=5]  $\rho$ )

# Language Specification

- Recursion
  - letrec fact(n) = if zero?(n) then 1 else \*(n, (fact {(n, 1)}))
  - in (fact 5)
- (value-of (letrec-exp p-names params p-bodys letrec-body)  $\rho$ ) = ???
- = (value-of letrec-body ???)
- What env is needed to evaluate the body?
- We need an env that contains a binding for all the local functions defined
- Create the procedure first?
  - (value-of
    - (fact 5)
      - [fact=(procedure ' (n) (if zero?(n) then 1 else \*(n, (fact {(n, 1)})) ?)]  $\rho$ )
  - = (value-of
    - (fact 5)
      - [fact=(procedure
        - ' (n)
          - (if zero?(n) then 1 else \*(n, (fact {(n, 1)}))
        - $\rho$ )]
      - $\rho$ )
    - = (value-of
      - \* (n, (fact {(n, 1)}))
        - [n=5]
          - [fact=(procedure
            - ' (n)
              - (if zero?(n) then 1 else \*(n, (fact {(n, 1)}))
            - $\rho$ )]
          - $\rho$ )
        - = (value-of \*(n, (fact {(n, 1)})) [n=5]  $\rho$ )
        - $\vdots$

# Language Specification

- Recursion
  - letrec fact(n) = if zero?(n) then 1 else \*(n, (fact {(n, 1)}))
  - in (fact 5)
- (value-of (letrec-exp p-names params p-bodys letrec-body)  $\rho$ ) = ???
- = (value-of letrec-body ???)
- What env is needed to evaluate the body?
- We need an env that contains a binding for all the local functions defined
- Create the procedure first?
  - (value-of
    - (fact 5)
    - [fact=(procedure ' (n) (if zero?(n) then 1 else \*(n, (fact {(n, 1)}))) ?]  $\rho$ )
- = (value-of
  - (fact 5)
  - [fact=(procedure
    - ' (n)
    - (if zero?(n) then 1 else \*(n, (fact {(n, 1)})))
    - $\rho$ ]
  - $\rho$ )
- = (value-of
  - \* (n, (fact {(n, 1)}))
  - [n=5]
  - [fact=(procedure
    - ' (n)
    - (if zero?(n) then 1 else \*(n, (fact {(n, 1)})))
    - $\rho$ ]
  - $\rho$ )
- = (value-of \*(n, (fact {(n, 1)})) [n=5]  $\rho$ )
- $\vdots$
- = (value-of (fact {(n, 1)}) [n=5]  $\rho$ )

# Language Specification

- Recursion
  - letrec fact(n) = if zero?(n) then 1 else \*(n, (fact {(n, 1)}))
  - in (fact 5)
- (value-of (letrec-exp p-names params p-bodys letrec-body)  $\rho$ ) = ???
- = (value-of letrec-body ???)
- What env is needed to evaluate the body?
- We need an env that contains a binding for all the local functions defined
- Create the procedure first?
  - (value-of
    - (fact 5)
    - [fact=(procedure '(n) (if zero?(n) then 1 else \*(n, (fact {(n, 1)}))) ?)]  $\rho$ )
- = (value-of
  - (fact 5)
  - [fact=(procedure
    - '(n)
    - (if zero?(n) then 1 else \*(n, (fact {(n, 1)})))
    - $\rho$ ]]
  - $\rho$ )
- = (value-of
  - \* (n, (fact {(n, 1)}))
  - [n=5]
  - [fact=(procedure
    - '(n)
    - (if zero?(n) then 1 else \*(n, (fact {(n, 1)})))
    - $\rho$ ]]
    - $\rho$ )
- = (value-of \*(n, (fact {(n, 1)})) [n=5]  $\rho$ )
- $\vdots$
- = (value-of (fact {(n, 1)}) [n=5]  $\rho$ )
- = error fact is not in the env; We can't create the procedure first

# Language Specification

- Recursion  
letrec fact(n) = if zero?(n) then 1 else \*(n, (fact (n, 1)))  
in (fact 5)
- (value-of letrec-exp(p-names params p-body's letrec-body)  $\rho$ ) = ???
- = (value-of letrec-body ???)
- What env is needed to evaluate the body?
- We need an env that contains a binding for all the local functions defined
- Create the needed env first?  
[fact=?] $\rho$ ]

# Language Specification

- Recursion  
letrec fact(n) = if zero?(n) then 1 else \*(n, (fact (n, 1)))  
in (fact 5)
- (value-of letrec-exp(p-names params p-bodys letrec-body)  $\rho$ ) = ???
- = (value-of letrec-body ???)
- What env is needed to evaluate the body?
- We need an env that contains a binding for all the local functions defined
- Create the needed env first?  
[fact=?]  $\rho$ ]
- Create the needed env first?  
[fact=(procedure ... ... ?)]  $\rho$ ]

# Language Specification

- Recursion  
letrec fact(n) = if zero?(n) then 1 else \*(n, (fact {(n, 1)}))  
in (fact 5)
- (value-of letrec-exp(p-names params p-bodys letrec-body)  $\rho$ ) = ???
- = (value-of letrec-body ???)
- What env is needed to evaluate the body?
- We need an env that contains a binding for all the local functions defined
- Create the needed env first?  
[fact=?]  $\rho$ ]
- Create the needed env first?  
[fact=(procedure ... ... ?)]  $\rho$ ]
- [fact=(procedure ... ... [fact=...]  $\rho$ )]  $\rho$ ]



# Language Specification

- Recursion  
letrec fact(n) = if zero?(n) then 1 else \*(n, (fact {(n, 1)}))  
in (fact 5)
- (value-of letrec-exp(p-names params p-bodys letrec-body)  $\rho$ ) = ???
- = (value-of letrec-body ???)
- What env is needed to evaluate the body?
- We need an env that contains a binding for all the local functions defined
- Create the needed env first?  
[fact=?]  $\rho$ ]
- Create the needed env first?  
[fact=(procedure ... ... ?)]  $\rho$ ]
- [fact=(procedure ... ... [fact=...]  $\rho$ )]  $\rho$ ]
- The required env needs the procedure for fact
- The procedure for fact needs the required env
- What is this known as?

# Language Specification

- Recursion  
letrec fact(n) = if zero?(n) then 1 else \*(n, (fact {(n, 1)}))  
in (fact 5)
- (value-of letrec-exp(p-names params p-bodys letrec-body)  $\rho$ ) = ???  
= (value-of letrec-body ???)
- What env is needed to evaluate the body?
- We need an env that contains a binding for all the local functions defined
- Create the needed env first?  
[fact=?]  $\rho$ ]
- Create the needed env first?  
[fact=(procedure ... ... ?)]  $\rho$ ]
- [fact=(procedure ... ... [fact=...]  $\rho$ )]  $\rho$ ]
- The required env needs the procedure for fact
- The procedure for fact needs the required env
- What is this known as?
- The Chicken and the Egg Paradox
- How do we solve it?

# Language Specification

- Recursion  
`letrec fact(n) = if zero?(n) then 1 else *(n, (fact (n, 1)))`  
`in (fact 5)`
- $(\text{value-of letrec-exp}(p\text{-names } p\text{-bodys letrec-body}) \rho) = ???$
- $= (\text{value-of letrec-body } ???)$
- What env is needed to evaluate the body?
- We need an env that contains a binding for all the local functions defined
- Create the needed env first?  
`[fact=?] ρ]`
- Create the needed env first?  
`[fact=(procedure ... ... ?)] ρ]`
- `[fact=(procedure ... ... [fact=...] ρ)] ρ]`
- The required env needs the procedure for fact
- The procedure for fact needs the required env
- What is this known as?
- The Chicken and the Egg Paradox
- How do we solve it?

## 97 Solving the Paradox

Given that neither a client nor an account can be created first using the constructors for the respective structures, a new type of constructor is needed. A *generalized constructor* builds incorrect structure instances. Mutation is used to correct the values in the instances. As problem-solvers, we need to decide which structure type is returned by a general constructor.

A generalized constructor may be written for any structure in a circular dependency, and later fields are mutated to correct the structure instances. For example, to build a **client**, the client's name and the initial balance of the first account may be used to build a **client** with no accounts. Observe that this violates the data definition for a **client**. Later the client's list of accounts is mutated to add the first account. We now proceed to design and implement a generalized constructor for **clients**.

# Language Specification

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

- We need an env that contains a binding for all the local functions defined
- $$\frac{\rho_r = [n1 = (\text{procedure } p1 \text{ } b1 \text{ } \rho_r)] \dots [nn = (\text{procedure } pn \text{ } bn \text{ } \rho_r)] \rho}{(\text{value-of } (\text{letrec-exp } (n1 \dots nn) (p1 \dots pn) (b1 \dots bn) \text{ body}) \rho) = (\text{value-of } \text{body } \rho_r)}$$

# Implementation

- Procedure representation

```
(define-datatype proc proc?
 (procedure
 (var (list-of symbol?))
 (body expression?)
 (envv voenv?)))
```

# Implementation

- Procedure representation

```
(define-datatype proc proc?
 (procedure
 (var (list-of symbol?))
 (body expression?)
 (envv voenv?)))
```

- ;; Any  $\rightarrow$  Boolean

```
;; Purpose: Determine if given value is a vector with a single environment
(define (voenv? penv)
 (and (vector? penv)
 (= (vector-length penv) 1)
 (environment? (vector-ref penv 0))))
```

# Implementation

- ```
;; value-of-program : program → expval
;; Purpose: Evaluate the given program
(define (value-of-program pgm)
  (cases program pgm
    (a-program (exp1)
      (value-of exp1 (empty-env))))))
```

Implementation

- ```
;; value-of-program : program → expval
;; Purpose: Evaluate the given program
(define (value-of-program pgm)
 (cases program pgm
 (a-program (exp1)
 (value-of exp1 (empty-env))))))
```
- ```
;; value-of : expression env → expval
;; Purpose: Evaluate the given expression in the given env
(define (value-of exp env)
  (cases expression exp
    (const-exp (num) ...)
    (true-exp () ...)
    (false-exp () ...)
    (var-exp (var) ...)))
```


Implementation

- ```
;; value-of-program : program → expval
;; Purpose: Evaluate the given program
(define (value-of-program pgm)
 (cases program pgm
 (a-program (exp1)
 (value-of exp1 (empty-env))))))
```
- ```
;; value-of : expression env → expval
;; Purpose: Evaluate the given expression in the given env
(define (value-of exp env)
  (cases expression exp
    (const-exp (num) ...)
    (true-exp () ...)
    (false-exp () ...)
    (var-exp (var) ...)
    (diff-exp (exp1 exp2) ... (value-of exp1 env) (value-of exp2 env))
    (zero?-exp (exp1) ... (value-of exp1 env))
    (if-exp (exp1 exp2 exp3) ...
      (value-of exp1 env)
      (value-of exp2 env)
      (value-of exp3 env))))
```

Implementation

- ```
;; value-of-program : program → expval
;; Purpose: Evaluate the given program
(define (value-of-program pgm)
 (cases program pgm
 (a-program (exp1)
 (value-of exp1 (empty-env))))))
```
- ```
;; value-of : expression env → expval
;; Purpose: Evaluate the given expression in the given env
(define (value-of exp env)
  (cases expression exp
    (const-exp (num) ...)
    (true-exp () ...)
    (false-exp () ...)
    (var-exp (var) ...)
```
- ```
 (diff-exp (exp1 exp2) ... (value-of exp1 env) (value-of exp2 env))
 (zero?-exp (exp1) ... (value-of exp1 env))
 (if-exp (exp1 exp2 exp3) ...
 (value-of exp1 env)
 (value-of exp2 env)
 (value-of exp3 env))
```
- ```
    (let-exp (vars exps body) ...
      (map (lambda (e) (value-of e env)) exps)
      (value-of body ...))
```

Implementation

- ```
;; value-of-program : program → expval
;; Purpose: Evaluate the given program
(define (value-of-program pgm)
 (cases program pgm
 (a-program (exp1)
 (value-of exp1 (empty-env))))))
```
- ```
;; value-of : expression env → expval
;; Purpose: Evaluate the given expression in the given env
(define (value-of exp env)
  (cases expression exp
    (const-exp (num) ...)
    (true-exp () ...)
    (false-exp () ...)
    (var-exp (var) ...)
```
- ```
(diff-exp (exp1 exp2) ... (value-of exp1 env) (value-of exp2 env))
(zero?-exp (exp1) ... (value-of exp1 env))
(if-exp (exp1 exp2 exp3) ...
 (value-of exp1 env)
 (value-of exp2 env)
 (value-of exp3 env))
```
- ```
(let-exp (vars exps body) ...
      (map (lambda (e) (value-of e env)) exps)
      (value-of body ...))
```
- ```
(proc-exp (params body) ...)
(call-exp (rator rands) ...
 (value-of rator env)
 (map (lambda (rand) (value-of rand env)) rands))
```

# Implementation

- ```
;; value-of-program : program → expval
;; Purpose: Evaluate the given program
(define (value-of-program pgm)
  (cases program pgm
    (a-program (exp1)
      (value-of exp1 (empty-env))))))
```
- ```
;; value-of : expression env → expval
;; Purpose: Evaluate the given expression in the given env
(define (value-of exp env)
 (cases expression exp
 (const-exp (num) ...)
 (true-exp () ...)
 (false-exp () ...)
 (var-exp (var) ...)
 (diff-exp (exp1 exp2) ... (value-of exp1 env) (value-of exp2 env))
 (zero?-exp (exp1) ... (value-of exp1 env))
 (if-exp (exp1 exp2 exp3) ...
 (value-of exp1 env)
 (value-of exp2 env)
 (value-of exp3 env))
 (let-exp (vars exps body) ...
 (map (lambda (e) (value-of e env)) exps)
 (value-of body ...))
 (proc-exp (params body) ...)
 (call-exp (rator rands) ...
 (value-of rator env)
 (map (lambda (rand) (value-of rand env)) rands))
 (letrec-exp (p-names params p-bodys letrec-body) (value-of letrec-body ...))))
```

# Implementation

- Specializing value-of

# Implementation

- Specializing value-of
- `(const-exp (num) (num-val num))`

# Implementation

- Specializing `value-of`
- `(const-exp (num) (num-val num))`
- $(\text{value-of } (\text{true-exp } n) \rho) = (\text{bool-val } \#t)$
- $(\text{value-of } (\text{false-exp } n) \rho) = (\text{bool-val } \#f)$

# Implementation

- Specializing value-of
- `(const-exp (num) (num-val num))`
- $(\text{value-of } (\text{true-exp } n) \rho) = (\text{bool-val } \#t)$
- $(\text{value-of } (\text{false-exp } n) \rho) = (\text{bool-val } \#f)$
- `(var-exp (var) (apply-env env var))`



# Implementation

- $$\frac{v1=(\text{expval2num } (\text{value-of } e1 \ \rho)) \ \wedge \ v2=(\text{expval2num } (\text{value-of } e2 \ \rho))}{(\text{diff-exp } e1 \ e2) = (\text{num-val } (- \ v1 \ v2))}$$

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

**Implementation**

Variable  
Names  
Elimination

# Implementation

- $$\frac{v1=(\text{expval2num } (\text{value-of } e1 \ \rho)) \ \wedge \ v2=(\text{expval2num } (\text{value-of } e2 \ \rho))}{(\text{diff-exp } e1 \ e2) = (\text{num-val } (- \ v1 \ v2))}$$
- ```
(diff-exp (exp1 exp2)
  (let ((num1 (expval2num (value-of exp1 env)))
        (num2 (expval2num (value-of exp2 env))))
    (num-val (- num1 num2))))
```

Implementation

- $$\frac{v1=(\text{expval2num } (\text{value-of } e1 \ \rho)) \ \wedge \ v2=(\text{expval2num } (\text{value-of } e2 \ \rho))}{(\text{diff-exp } e1 \ e2) = (\text{num-val } (- \ v1 \ v2))}$$
- $$(\text{diff-exp } (\text{exp1 } \text{exp2}))$$

$$(\text{let } ((\text{num1 } (\text{expval2num } (\text{value-of } \text{exp1 } \text{env})))$$

$$(\text{num2 } (\text{expval2num } (\text{value-of } \text{exp2 } \text{env}))))$$

$$(\text{num-val } (- \ \text{num1 } \text{num2}))))$$
- $$\frac{v=(\text{expval2num } (\text{value-of } e) \ \rho)}{(\text{value-of } (\text{zero?-exp } n) \ \rho) = \begin{cases} (\text{bool-val } \#t), & \text{if } v = 0 \\ (\text{bool-val } \#f), & \text{if } v \neq 0 \end{cases}}$$

Implementation

- $$\frac{v1=(\text{expval2num } (\text{value-of } e1 \ \rho)) \ \wedge \ v2=(\text{expval2num } (\text{value-of } e2 \ \rho))}{(\text{diff-exp } e1 \ e2) = (\text{num-val } (- \ v1 \ v2))}$$
- ```
(diff-exp (exp1 exp2)
 (let ((num1 (expval2num (value-of exp1 env)))
 (num2 (expval2num (value-of exp2 env))))
 (num-val (- num1 num2))))
```
- $$\frac{v=(\text{expval2num } (\text{value-of } e) \ \rho)}{(\text{value-of } (\text{zero?-exp } n) \ \rho) = \begin{cases} (\text{bool-val } \#t), & \text{if } v = 0 \\ (\text{bool-val } \#f), & \text{if } v \neq 0 \end{cases}}$$
- ```
(zero?-exp (exp1)
  (let ((val1 (expval2num (value-of exp1 env))))
    (if (zero? val1)
        (bool-val #t)
        (bool-val #f))))
```

Implementation

- $$\frac{v1=(\text{expval2num } (\text{value-of } e1 \ \rho)) \ \wedge \ v2=(\text{expval2num } (\text{value-of } e2 \ \rho))}{(\text{diff-exp } e1 \ e2) = (\text{num-val } (- \ v1 \ v2))}$$
- $$(\text{diff-exp } (\text{exp1 } \text{exp2}))$$

```
(let ((num1 (expval2num (value-of exp1 env)))
      (num2 (expval2num (value-of exp2 env))))
  (num-val (- num1 num2)))
```
- $$\frac{v=(\text{expval2num } (\text{value-of } e \ \rho))}{(\text{value-of } (\text{zero?-exp } n) \ \rho) = \begin{cases} (\text{bool-val } \#t), & \text{if } v = 0 \\ (\text{bool-val } \#f), & \text{if } v \neq 0 \end{cases}}$$
- $$(\text{zero?-exp } (\text{exp1}))$$

```
(let ((val1 (expval2num (value-of exp1 env))))
  (if (zero? val1)
      (bool-val #t)
      (bool-val #f)))
```
- $$\frac{cval=(\text{expval2bool } (\text{value-of } c \ \rho))}{(\text{value-of } (\text{if-exp } c \ t \ e) \ \rho) = \begin{cases} (\text{value-of } t \ \rho), & \text{if } cval = \#t \\ (\text{value-of } e \ \rho), & \text{if } cval = \#f \end{cases}}$$

Implementation

- $$\frac{v1=(\text{expval2num } (\text{value-of } e1 \ \rho)) \ \wedge \ v2=(\text{expval2num } (\text{value-of } e2 \ \rho))}{(\text{diff-exp } e1 \ e2) = (\text{num-val } (- \ v1 \ v2))}$$
- ```
(diff-exp exp1 exp2)
 (let ((num1 (expval2num (value-of exp1 env)))
 (num2 (expval2num (value-of exp2 env))))
 (num-val (- num1 num2))))
```
- $$\frac{v=(\text{expval2num } (\text{value-of } e \ \rho))}{(\text{value-of } (\text{zero?-exp } n) \ \rho) = \begin{cases} (\text{bool-val } \#t), & \text{if } v = 0 \\ (\text{bool-val } \#f), & \text{if } v \neq 0 \end{cases}}$$
- ```
(zero?-exp exp1)
  (let ((val1 (expval2num (value-of exp1 env))))
    (if (zero? val1)
        (bool-val #t)
        (bool-val #f))))
```
- $$\frac{cval=(\text{expval2bool } (\text{value-of } c \ \rho))}{(\text{value-of } (\text{if-exp } c \ t \ e) \ \rho) = \begin{cases} (\text{value-of } t \ \rho), & \text{if } cval = \#t \\ (\text{value-of } e \ \rho), & \text{if } cval = \#f \end{cases}}$$
- ```
(if-exp exp1 exp2 exp3)
 (let ((val1 (value-of exp1 env)))
 (if (expval2bool val1)
 (value-of exp2 env)
 (value-of exp3 env))))
```

# Implementation

- $$\frac{(v1 \dots vn) = (\text{map } (\text{lambda } (e) (\text{value-of } e \ \rho)) (e1 \dots en))}{(\text{value-of } (\text{let-exp } (s1 \dots sn) (e1 \dots en) \text{ body}) \rho) = (\text{value-of } \text{body } [s1=v1 \dots sn=vn] \rho)}$$

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

# Implementation

- $$\frac{(v1...vn)=(\text{map } (\text{lambda } (e) (\text{value-of } e \rho))(e1...en))}{(\text{value-of } (\text{let-exp } (s1...sn) (e1...en) \text{ body}) \rho) = (\text{value-of } \text{body } [s1=v1...sn=vn] \rho)}$$
- ```
(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))))
```


Implementation

- $$\frac{(v1...vn)=(\text{map } (\text{lambda } (e) (\text{value-of } e \ \rho)) (e1...en))}{(\text{value-of } (\text{let-exp } (s1...sn) (e1...en) \text{ body}) \rho) = (\text{value-of } \text{body } [s1=v1...sn=vn] \rho)}$$
- ```

(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))))

```
- $$(\text{value-of } (\text{proc-exp } (p1...pn) b) \rho) = (\text{proc-val } (\text{procedure } (p1...pn) b \rho))$$

# Implementation

- $$\frac{(v1...vn)=(\text{map } (\text{lambda } (e) (\text{value-of } e \ \rho)) (e1...en))}{(\text{value-of } (\text{let-exp } (s1...sn) (e1...en) \text{ body}) \rho) = (\text{value-of } \text{body } [s1=v1...sn=vn] \rho)}$$
- ```

(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))))

```
- $$(\text{value-of } (\text{proc-exp } (p1...pn) b) \rho) = (\text{proc-val } (\text{procedure } (p1...pn) b \rho))$$
- ```

(proc-exp (params body)
 (proc-val (procedure params body (vector env))))

```

# Implementation

- $$\frac{(v1...vn)=(\text{map } (\text{lambda } (e) (\text{value-of } e \ \rho)) (e1...en))}{(\text{value-of } (\text{let-exp } (s1...sn) (e1...en) \text{ body}) \rho) = (\text{value-of } \text{body } [s1=v1...sn=\rho])}$$
- ```
(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))))
```
- ```
(value-of (proc-exp (p1...pn) b) ρ) = (proc-val (procedure (p1...pn) b ρ))
```
- ```
(proc-exp (params body)
  (proc-val (procedure params body (vector env))))
```
- $$\frac{p=(\text{expval} \rightarrow \text{proc}(\text{value-of } e0 \ \rho)) \wedge \text{args} = (\text{map } (\text{lambda } (e) (\text{value-of } e \ \text{env})) (e1...en))}{(\text{value-of } (\text{call-exp } e0 (e1...en)) \rho) = (\text{apply-procedure } p \ \text{args})}$$

Implementation

- $$\frac{(v1...vn)=(\text{map } (\text{lambda } (e) (\text{value-of } e \ \rho))(e1...en))}{(\text{value-of } (\text{let-exp } (s1...sn) (e1...en) \text{ body}) \rho) = (\text{value-of } \text{body } [s1=v1...sn=vn] \rho)}$$
- ```
(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))))
```
- ```
(value-of (proc-exp (p1...pn) b) ρ) = (proc-val (procedure (p1...pn) b ρ))
```
- ```
(proc-exp (params body)
 (proc-val (procedure params body (vector env))))
```
- $$\frac{p=(\text{expval} \rightarrow \text{proc}(\text{value-of } e0 \ \rho)) \wedge \text{args} = (\text{map } (\text{lambda } (e) (\text{value-of } e \ \text{env})) (e1...en))}{(\text{value-of } (\text{call-exp } e0 (e1...en)) \rho) = (\text{apply-procedure } p \ \text{args})}$$
- ```
(call-exp (rator rands)
  (let [(proc (expval2proc (value-of rator env))]
    (args (map (lambda (rand) (value-of rand env)) rands))]
    (apply-procedure proc args)))
```

Implementation

- ```
;; apply-procedure : proc (listof expval) → expval
;; Purpose: Apply the given procedure to the given values
(define (apply-procedure f vals)
 (cases proc f
 (procedure (params body envv)
 (let [(saved-env (vector-ref envv 0))]
 (value-of body
 (foldr (lambda (binding acc)
 (extend-env (car binding)
 (cadr binding)
 acc))
 saved-env
 (map (lambda (p v) (list p v))
 params
 vals)))))))
```

# Implementation

- $$\frac{\rho_r = [n1 = (\text{procedure } p1 \text{ } b1 \text{ } \rho_r)] \dots [nn = (\text{procedure } pn \text{ } bn \text{ } \rho_r)] \rho}{(\text{value-of } (\text{letrec-exp } (n1 \dots nn) (p1 \dots pn) (b1 \dots bn) \text{ body}) \rho) = (\text{value-of } \text{body } \rho_r)}$$

# Implementation

$$\bullet \frac{\rho_r = [n1 = (\text{procedure } p1 \text{ } b1 \text{ } \rho_r)] \dots [nn = (\text{procedure } pn \text{ } bn \text{ } \rho_r)] \rho}{(\text{value-of } (\text{letrec-exp } (n1 \dots nn) (p1 \dots pn) (b1 \dots bn) \text{ body}) \rho) = (\text{value-of } \text{body } \rho_r)}$$

•

```
;; (listof symbol) (listof (listof symbol)) (listof expression) env
```

```
;; Purpose: Add proc-vals for given procedures in given environment
```

```
(define (mk-letrec-env ns ps bs env) Generalized constructor
```

```
 (let* [(temp-proc-vals
```

```
 (map (lambda (p b)
```

**Temporary wrong proc-vals**

```
 (proc-val (procedure p b (vector (empty-env))))))
```

```
 ps
```

```
 bs))
```

# Implementation

- $$\frac{\rho_r = [n1 = (\text{procedure } p1 \text{ } b1 \text{ } \rho_r)] \dots [nn = (\text{procedure } pn \text{ } bn \text{ } \rho_r)] \rho}{(\text{value-of } (\text{letrec-exp } (n1 \dots nn) (p1 \dots pn) (b1 \dots bn) \text{ body}) \rho) = (\text{value-of } \text{body } \rho_r)}$$

- ```
;; (listof symbol) (listof (listof symbol)) (listof expression) env
;; Purpose: Add proc-vals for given procedures in given environment
(define (mk-letrec-env ns ps bs env) Generalized constructor
  (let* [(temp-proc-vals
          (map (lambda (p b)
                 (proc-val (procedure p b (vector (empty-env))))))
          ps
          bs))
        (new-env (new-env (foldl (lambda (name proc env)
                                   (extend-env name proc env))
                                env
                                names
                                temp-proc-vals))))]
```

Temporary wrong proc-vals

Implementation

$$\frac{\rho_r = [n1 = (\text{procedure } p1 \text{ } b1 \text{ } \rho_r)] \dots [nn = (\text{procedure } pn \text{ } bn \text{ } \rho_r)] \rho}{(\text{value-of } (\text{letrec-exp } (n1 \dots nn) (p1 \dots pn) (b1 \dots bn) \text{ body}) \rho) = (\text{value-of } \text{body } \rho_r)}$$

- ```
;; (listof symbol) (listof (listof symbol)) (listof expression) env
;; Purpose: Add proc-vals for given procedures in given environment
(define (mk-letrec-env ns ps bs env) Generalized constructor
 (let* [(temp-proc-vals
 (map (lambda (p b)
 (proc-val (procedure p b (vector (empty-env))))))
 ps
 bs))
 (new-env (new-env (foldl (lambda (name proc env)
 (extend-env name proc env))
 env
 names
 temp-proc-vals)))]
```
- ```
(begin
  (for-each (lambda (p) Correcting proc-vals
    (cases proc p
      (procedure (p b ve) (vector-set! ve 0 new-env)))
    (map (lambda (p) (expval2proc p)) temp-proc-vals))
```

Implementation

$$\frac{\rho_r = [n1 = (\text{procedure } p1 \text{ } b1 \text{ } \rho_r)] \dots [nn = (\text{procedure } pn \text{ } bn \text{ } \rho_r)] \rho}{(\text{value-of } (\text{letrec-exp } (n1 \dots nn) (p1 \dots pn) (b1 \dots bn) \text{ body}) \rho) = (\text{value-of } \text{body } \rho_r)}$$

- ;; (listof symbol) (listof (listof symbol)) (listof expression) env
;; Purpose: Add proc-vals for given procedures in given environment
(define (mk-letrec-env ns ps bs env) **Generalized constructor**
 (let* [(temp-proc-vals
 (map (lambda (p b) **Temporary wrong proc-vals**
 (proc-val (procedure p b (vector (empty-env))))))
 ps
 bs))
 (new-env (new-env (foldl (lambda (name proc env)
 (extend-env name proc env))
 env
 names
 temp-proc-vals))))])
- (begin
 (for-each (lambda (p) **Correcting proc-vals**
 (cases proc p
 (procedure (p b ve) (vector-set! ve 0 new-env))))
 (map (lambda (p) (expval2proc p)) temp-proc-vals))
 new-env)))

Implementation

HOMEWORK

- Problems: 3.6–3.10, 3.12, 3.16–3.17, 3.21, 3.23–3.24, 3.26, 3.32–3.33, 3.55 (using the interpreter developed in class)
- Some problems we have already solved!

Variable Names Elimination

- Several ways to declare vars: let, letrec, and proc (so far!)

A Language
Based on an
Extended
Lambda
Calculus

Evaluation
Wrapper
Functions and
Tests

Expressed and
Denoted
Values

Environment
Datatype

Language
Specification

Implementation

Variable
Names
Elimination

Variable Names Elimination

- Several ways to declare vars: let, letrec, and proc (so far!)

- In most PLs, declarations have limited scope

```
(let [(pi 3.14)  
      (e 2.71)]
```

```
(+ (let [(pi 3.4)]      (+ pi e) )    pi) )
```

- Every programming language must have *scoping rules*
- Rules for determining the declaration for a variable reference

Variable Names Elimination

- Several ways to declare vars: let, letrec, and proc (so far!)

- In most PLs, declarations have limited scope

```
(let [(pi 3.14)  
      (e 2.71)]
```

```
(+ (let [(pi 3.4)]      (+ pi e) )    pi) )
```

- Every programming language must have *scoping rules*
- Rules for determining the declaration for a variable reference
- In many PLs, search outward from the reference to the declaration
- This is called lexical scoping and it is a static property

Variable Names Elimination

- `(let [(pi 3.14)
 (e 2.71)]`

`(+ (let [(pi 3.4)] (+ pi e)) pi))`

- Holes in the scope of a var may be created by nested declarations
- The number of boxes crossed is the lexical depth of a var
- The position in the declarations in the lexical position
- Lexical address is both the lexical depth and lexical position

Variable Names Elimination

- `(let [(pi 3.14)
 (e 2.71)]`

`(+ (let [(pi 3.4)] (+ pi e)) pi))`

- Holes in the scope of a var may be created by nested declarations
- The number of boxes crossed is the lexical depth of a var
- The position in the declarations in the lexical position
- Lexical address is both the lexical depth and lexical position
- `pi`: 0 0
- `e`: 1 1
- `pi`: 0 0

Variable Names Elimination

- `(let [(pi 3.14)
 (e 2.71)]`

```
(+ (let [(pi 3.4)]      (+ pi e) )   pi) )
```

- Holes in the scope of a var may be created by nested declarations
- The number of boxes crossed is the lexical depth of a var
- The position in the declarations in the lexical position
- Lexical address is both the lexical depth and lexical position
- `pi`: 0 0
- `e`: 1 1
- `pi`: 0 0
- Why is this important?

Variable Names Elimination

- `(let [(pi 3.14)
 (e 2.71)]`

`(+ (let [(pi 3.4)] (+ pi e)) pi))`

- Holes in the scope of a var may be created by nested declarations
- The number of boxes crossed is the lexical depth of a var
- The position in the declarations in the lexical position
- Lexical address is both the lexical depth and lexical position
- `pi`: 0 0
- `e`: 1 1
- `pi`: 0 0
- Why is this important?
- Using lexical addresses eliminates the need to search for a binding in an environment.

Variable Names Elimination

- ```
(let [(pi 3.14)
 (e 2.71)]
 (+ (let [(pi 3.4)]
 (+ pi e))
 pi))
```

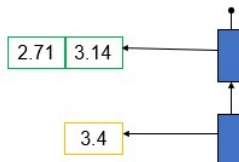
# Variable Names Elimination

- ```
(let [(pi 3.14)
      (e 2.71)]
  (+ (let [(pi 3.4)]
      (+ pi e))
     pi))
```
- Implement an env as a list of ribs, where a rib is a list of expvals

Variable Names Elimination

- ```
(let [(pi 3.14)
 (e 2.71)]
 (+ (let [(pi 3.4)]
 (+ pi e))
 pi))
```

- Implement an env as a list of ribs, where a rib is a list of expvals



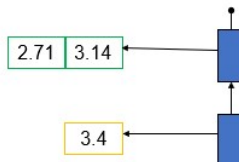
•

# Variable Names Elimination

- `(let [(pi 3.14)  
      (e 2.71)]`

```
(+ (let [(pi 3.4)] (+ pi e)) pi))
```

- Implement an env as a list of ribs, where a rib is a list of expvals



- `(let [(3.14 2.71)]`

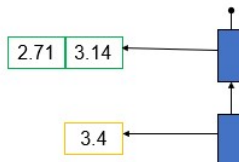
```
(+ (let [(pi 3.4)] (+ (0 0) (1 1))) (0 0)))
```

# Variable Names Elimination

- `(let [(pi 3.14)  
      (e 2.71)]`

```
(+ (let [(pi 3.4)] (+ pi e)) pi))
```

- Implement an env as a list of ribs, where a rib is a list of expvals



- `(let [(3.14 2.71)]`

```
(+ (let [(pi 3.4)] (+ (0 0) (1 1))) (0 0)))
```

- The same is done for: `proc`, `letrec`
- A variable becomes a lexical address
- Add nameless versions for `var`, `proc`, `let`, and `letrec` to extended LC grammar
- Change env representation

# Variable Names Elimination

- ;;;; ENVIRONMENT

#|

A rib is a (listof expval)

An environment is a (listof rib)

|#



# Variable Names Elimination

- `;;;;; ENVIRONMENT`  
  
`#|`  
  
`A rib is a (listof expval)`  
  
`An environment is a (listof rib)`  
  
`|#`
- `(define (environment? e)`  
    `(list-of (list-of expval?)))`

# Variable Names Elimination

- `;;;; ENVIRONMENT`  
  
`#|`  
  
`A rib is a (listof expval)`  
  
`An environment is a (listof rib)`  
  
`|#`
- `(define (environment? e)`  
    `(list-of (list-of expval?)))`
- `;; → environment`  
    `;; Purpose: Build the empty env`  
    `(define (empty-env) '())`

# Variable Names Elimination

- `;;;; ENVIRONMENT`  
  
`#|`  
  
`A rib is a (listof expval)`  
  
`An environment is a (listof rib)`  
  
`|#`
- `(define (environment? e)`  
    `(list-of (list-of expval?)))`
- `;; → environment`  
    `;; Purpose: Build the empty env`  
    `(define (empty-env) '())`
- `;; (listof expval) environment → environment`  
    `;; Purpose: Build an environment from given expvals and env`  
    `(define (extend-env vals env) (cons vals env))`

# Variable Names Elimination

- `;;;;; ENVIRONMENT`

`#|`

`A rib is a (listof expval)`

`An environment is a (listof rib)`

`|#`

- ```
(define (environment? e)
  (list-of (list-of expval?)))
```
- ```
;; → environment
;; Purpose: Build the empty env
(define (empty-env) '())
```
- ```
;; (listof expval) environment → environment
;; Purpose: Build an environment from given expvals and env
(define (extend-env vals env) (cons vals env))
```
- ```
;; environment natnum natnum → expval throws error
;; Purpose: Return expval at given lexical address in given env
(define (apply-env env depth pos)
 (if (empty? env)
 (eopl:error 'apply-env "No binding for lexical address: ~s ~s"
 (list-ref env depth) pos))
```

# Variable Names Elimination

- New grammar rules
- `(expression ("%lexvar" number number) nameless-var-exp)`

`(expression ("%nameless-proc" "(" expression "))" nameless-proc-exp)`

`(expression ("%let" (arbno expression) "in" expression) nameless-let-exp)`

`(expression  
 ("%letrec" (arbno expression) "in" expression) nameless-letrec-exp)`

# Variable Names Elimination

- procs no longer need to store the parameter names

```
(define-datatype proc proc?
 (procedure
 (body expression?)
 (envv voenv?)))
```

# Variable Names Elimination

A Language  
Based on an  
Extended  
Lambda  
Calculus

Evaluation  
Wrapper  
Functions and  
Tests

Expressed and  
Denoted  
Values

Environment  
Datatype

Language  
Specification

Implementation

Variable  
Names  
Elimination

- procs no longer need the store the parameter names

```
(define-datatype proc proc?
 (procedure
 (body expression?)
 (envv voenv?)))
```
- To evaluate we must translate a program to an equivalent nameless version

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

# Variable Names Elimination

- Refinements to value-of

```
(nameless-var-exp (d p) (apply-env env d p))
```

```
(nameless-let-exp (exps body)
 (let [(vals (map (lambda (e) (value-of e env)) exps))]
 (value-of body (extend-env vals env))))
```

```
(nameless-proc-exp (body)
 (proc-val (procedure body (vector env))))
```

```
(nameless-letrec-exp (bodies letrec-body)
 (value-of letrec-body (mk-letrec-env bodies env)))
```



# Variable Names Elimination

## HOMEWORK

- Problems: 3.38, 3.41 (using the interpreter developed in class)