# Proc Review

T. METIN SEZGIN

# Implementation

```
proc? : SchemeVal → Bool
(define proc?
  (lambda (val)
    (procedure? val)))

procedure : Var × Exp × Env → Proc
(define procedure
  (lambda (var body env)
    (lambda (val)
      (value-of body (extend-env var val env)))))

apply-procedure : Proc × ExpVal → ExpVal
(define apply-procedure
  (lambda (proc1 val)
    (proc1 val)))
```

# Alternative implementation

```
proc?    :  SchemeVal  →  Bool
procedure  :  Var  ×  Exp  ×  Env  →  Proc
(define-datatype proc proc?
   (procedure
     (var identifier?)
     (body expression?)
     (saved-env environment?)))

apply-procedure  :  Proc  ×  ExpVal  →  ExpVal
(define apply-procedure
   (lambda (proc1 val)
     (cases proc proc1
        (procedure (var body saved-env)
          (value-of body (extend-env var val saved-env))))))
```

# Other changes to the interpreter

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


(proc-exp (var body)
  (proc-val (procedure var body env)))

(call-exp (rator rand)
  (let ((proc (expval->proc (value-of rator env)))
        (arg (value-of rand env)))
    (apply-procedure proc arg)))
```

# LETREC

T. METIN SEZGIN

# PROC is ex; long live LETREC

- PROC had its limitations
  - No recursive procedures
- Define a language with recursive procedures
  - Specification
    - Syntax
    - Semantics
  - Representation
  - Implementation

# Nuggets of the lecture

- Implementation requires creating representation for recursive procedures

- We need to rethink how we evaluate recursive procedures

- A more elaborate way of representing procedures in the environment is needed

# LETREC

- The idea

```
letrec double(x)
        = if zero?(x) then 0 else -((double -(x,1)), -2)
in (double 6)
```

- The new grammar

```
Expression ::= letrec Identifier (Identifier) = Expression in Expression
               letrec-exp (p-name b-var p-body letrec-body)
```

# LETREC

- Extend the environment recursively

```
(value-of
    (letrec-exp proc-name bound-var proc-body letrec-body)
    ρ)
= (value-of
      letrec-body
      (extend-env-rec proc-name bound-var proc-body ρ))
```
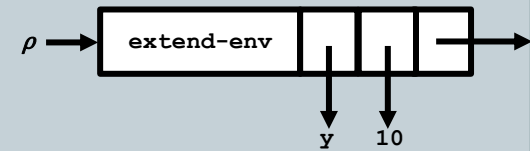
- How should environment lookup work?
  - If the search variable matches a recursive procedure

```
(apply-env ρ1 proc-name)
= (proc-val (procedure bound-var proc-body ρ1))
```
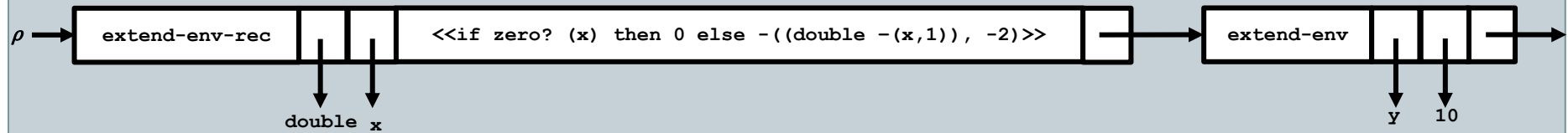
  - If there is no match

```
(apply-env ρ1 var) = (apply-env ρ var)
```

# Extended environment

$\rho$ → | **extend-env** | | | | → 

y   10

# Extended environment



```
ρ → | extend-env-rec | | | | | <<if zero? (x) then 0 else -((double -(x,1)), -2)>> | | → | extend-env | | | | | | | →
                      ↓   ↓                                                                              ↓   ↓
                   double  x                                                                             y   10
```

# Example

```
(value-of <<letrec double(x) = if zero?(x)
                                   then 0
                                   else -((double -(x,1)), -2)
               in (double 6)>> ρ₀)

= (value-of << (double 6)>>
     (extend-env-rec double x <<if zero?(x) ...>> ρ₀))

= (apply-procedure
     (value-of <<double>> (extend-env-rec double x
                                <<if zero?(x) ...>> ρ₀))
     (value-of <<6>> (extend-env-rec double x
                         <<if zero?(x) ...>> ρ₀)))

= (apply-procedure
     (procedure x <<if zero?(x) ...>>
        (extend-env-rec double x <<if zero?(x) ...>> ρ₀))
     ⌈6⌉)

= (value-of
     <<if zero?(x) ...>>
     [x=⌈6⌉] (extend-env-rec
                 double x <<if zero?(x) ...>> ρ₀))

...

= (-
     (value-of
      << (double -(x,1))>>
      [x=⌈6⌉] (extend-env-rec
                  double x <<if zero?(x) ...>> ρ₀))
     -2)
```

# Example cont.

```
=  (-
      (apply-procedure
        (value-of
          <<double>>
          [x=⌈6⌉] (extend-env-rec
                              double x <<if zero?(x)  ...>> ρ₀))
        (value-of
          <<-(x,1)>>
          [x=⌈6⌉] (extend-env-rec
                              double x <<if zero?(x)  ...>> ρ₀)))
    -2)

=  (-
      (apply-procedure
        (procedure x <<if zero?(x)  ...>>
          (extend-env-rec double x <<if zero?(x)  ...>> ρ₀))
        ⌈5⌉)
    -2)

= ...
```
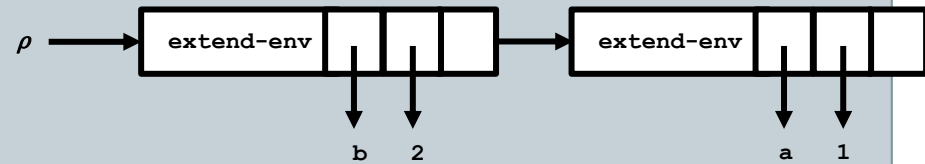
```
let a=1 in
  let b=2 in
    letrec f(x) = if zero?(x) then 0 else -((f -(x,1)),-2) in (f 2)")
```

```
let a=1 in
  let b=2 in
    letrec f(x) = if zero?(x) then 0 else -((f -(x,1)),-2) in (f 2)")
```



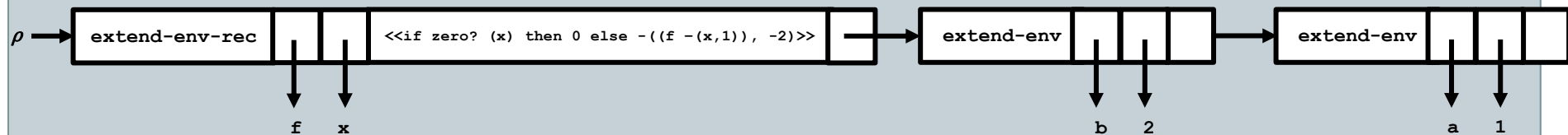ρ → extend-env

a   1

```
let a=1 in
  let b=2 in
    letrec f(x) = if zero?(x) then 0 else -((f -(x,1)),-2) in (f 2)")
```

```
let a=1 in
  let b=2 in
    letrec f(x) = if zero?(x) then 0 else -((f -(x,1)),-2) in (f 2)")
```

# The new **environment** and **apply-env**

```scheme
(define-datatype environment environment?
  (empty-env)
  (extend-env
    (var identifier?)
    (val expval?)
    (env environment?))
  (extend-env-rec
    (p-name identifier?)
    (b-var identifier?)
    (body expression?)
    (env environment?)))

(define apply-env
  (lambda (env search-var)
    (cases environment env
      (empty-env ()
        (report-no-binding-found search-var))
      (extend-env (saved-var saved-val saved-env)
        (if (eqv? saved-var search-var)
          saved-val
          (apply-env saved-env search-var)))
      (extend-env-rec (p-name b-var p-body saved-env)
        (if (eqv? search-var p-name)
          (proc-val (procedure b-var p-body env))
          (apply-env saved-env search-var)))))))
```