Part IV: State

Marco T. Morazán

Seton Hall University

Outline

State

- Computations may also have effects
  - print
  - change a memory location
  - change a file

State

- Computations may also have effects
  - print
  - change a memory location
  - change a file
- An effect is global: affects the entire computation

State

- Computations may also have effects
  - print
  - change a memory location
  - change a file
- An effect is global: affects the entire computation
- We will now study assignment (aka mutation)

State

- Assignment is about sharing values/information between unrelated parts of a computation
- CSAS 1115: telephone book, bank account

State

- Assignment is about sharing values/information between unrelated parts of a computation
- CSAS 1115: telephone book, bank account
- Memory model
  - A finite map of locations to storable values (aka store or heap)
  - A place in memory where values can be stored

State

- Assignment is about sharing values/information between unrelated parts of a computation
- CSAS 1115: telephone book, bank account
- Memory model
  - A finite map of locations to storable values (aka store or heap)
  - A place in memory where values can be stored
- Implementation
  - Typically, storable values are the same as the expressed values
  - A data structure that represents a location is called a reference (aka pointer)

State

- Two ways to design a language with store
- Explicit references
- programmer allocates, dereferences, and mutates locations/memory

State

- Two ways to design a language with store
- Explicit references
- programmer allocates, dereferences, and mutates locations/memory
- Implicit references
- language packages common patterns of allocation, dereferencing, and mutation

Language with Explicit References

- expval = int + bool + proc + *ref(expval)*
- denval = expval

Language with Explicit References

- expval = int + bool + proc + *ref(expval)*
- denval = expval
- 3 new ops needed:

newref	allocates a new location and returns a reference to the new location
deref	returns the content of a reference
setref	changes the content of a referenced location

Language with Explicit References

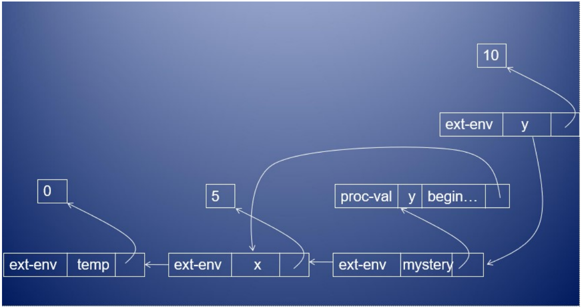
- Value?

```
let temp = newref(0)
in  let x = newref(5)
    in  let mystery = proc (y)
                        begin
                          setref(temp, deref(x));
                          setref(x, deref(y));
                          setref(y, deref(temp));
                          deref(x)
        end
    in (mystery newref(10))
```

Language with Explicit References

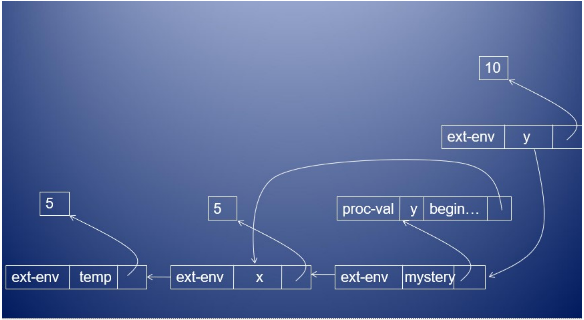
- Value?

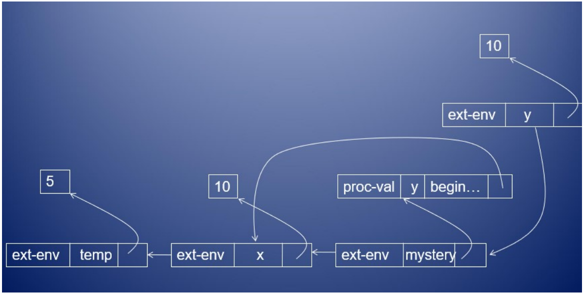
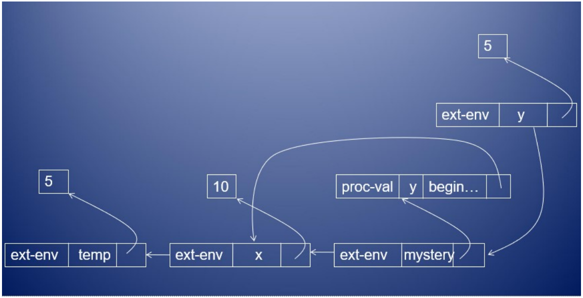
```
let temp = newref(0)
in  let x = newref(5)
    in  let mystery = proc (y)
                        begin
                          setref(temp, deref(x));
                          setref(x, deref(y));
                          setref(y, deref(temp));
                          deref(x)
        end
    in (mystery newref(10))
```

- 

Language with Explicit References

- ```
  let temp = newref(0)
  in  let x = newref(5)
      in  let mystery = proc (y)
                          begin
                            setref(temp, deref(x));
                            setref(x, deref(y));
                            setref(y, deref(temp));
                            deref(x)
          end
      in (mystery newref(10))
  ```



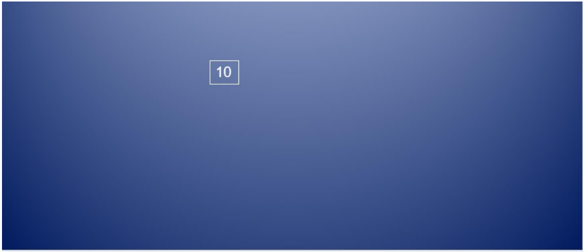
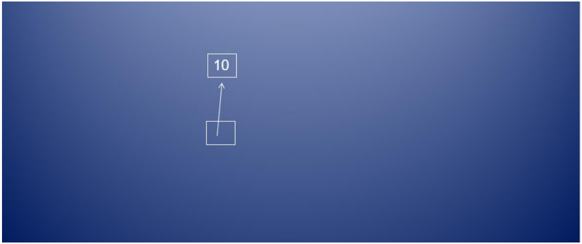
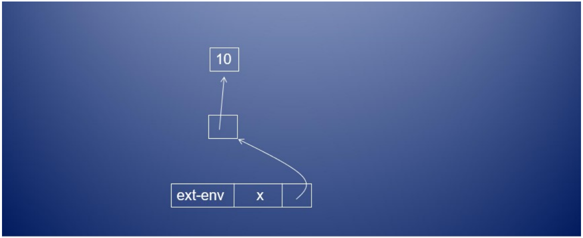
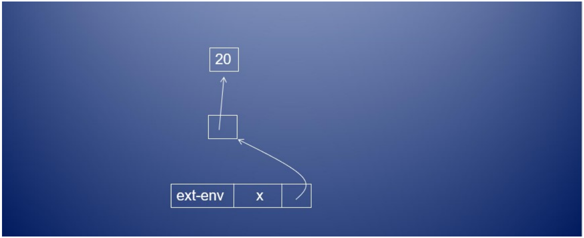
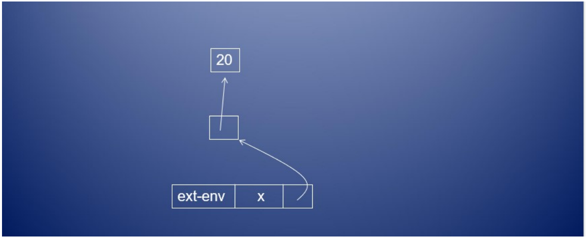
-
  ```

Language with Explicit References

```
let temp = newref(0)
in  let x = newref(5)
    in  let mystery = proc (y)
                        begin
                          setref(temp, deref(x));
                          setref(x, deref(y));
                          setref(y, deref(temp));
                          deref(x)
        end
    in (mystery newref(10))
```

- 

Language with Explicit References

- ```
  let temp = newref(0)
  in  let x = newref(5)
      in  let mystery = proc (y)
                          begin
                            setref(temp, deref(x));
                            setref(x, deref(y));
                            setref(y, deref(temp));
                            deref(x)
          end
      in (mystery newref(10))
  ```
- 

- Returns: 10
  ```

Language with Explicit References

- Is this a valid program? If so, what does it evaluate to?

```
let x = newref(newref(10))
in begin
setref(deref(x), 20));
 +(20, deref(deref(x)))
   end
```

Language with Explicit References

- ```
  let x = newref(newref(10))
  in begin
  setref(deref(x), 20));
   +(20, deref(deref(x)))
     end
  ```
- 
  ```

Language with Explicit References

- Is this a valid program? If so, what does it evaluate to?

```
let x = newref(newref(10))
in begin
setref(deref(x), 20));
 +(20, deref(deref(x)))
   end
```

- 

Language with Explicit References

- Is this a valid program? If so, what does it evaluate to?

```
let x = newref(newref(10))
in begin
setref(deref(x), 20));
 +(20, deref(deref(x)))
   end
```

- 

Language with Explicit References

- Is this a valid program? If so, what does it evaluate to?

```
let x = newref(newref(10))
in begin
setref(deref(x), 20));
+(20, deref(deref(x)))
   end
```

-


Language with Explicit References

- Is this a valid program? If so, what does it evaluate to?

```
let x = newref(newref(10))
in begin
setref(deref(x), 20));
+(20, deref(deref(x)))
   end
```

- 

- Returns 40

Language with Explicit References

- Expressed Values

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

Language with Explicit References

- Expressed Values

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

-
```
;; expval --> ref throws error
;; Purpose: Extract ref from given expval
(define (expval2ref v)
  (cases expval v
    (ref-val (ref) ref)
    (else (expval-extractor-error 'reference v))))
```

Language with Explicit References

- To have *effects* values must be stored somewhere

Language with Explicit References

- To have *effects* values must be stored somewhere
- σ ranges over the store (or heap)
- $[l = v]\sigma \rightarrow$ the store σ extended with location l mapped to v

Language with Explicit References

- To have *effects* values must be stored somewhere
- σ ranges over the store (or heap)
- $[l = v]\sigma \rightarrow$ the store σ extended with location l mapped to v
- We shall think of the store as an argument to value-of
- (value-of exp1 ρ σ_0) $=$ (val1, σ_1)
- σ_0 may or may not be the same as σ_1

Language with Explicit References

Specification

- (value-of (const-exp n) ρ σ) = ((numval n), σ)

Language with Explicit References

Specification

- (value-of (const-exp n) ρ σ) $=$ ((numval n), σ)
-
$$\frac{(\mathit{value\text{-}of}\ \ \mathit{exp}1\ \ \rho\ \ \sigma_0) = (\mathit{val}1,\ \sigma_1) \wedge (\mathit{value\text{-}of}\ \ \mathit{exp}2\ \ \rho\ \ \sigma_1) = (\mathit{val}2,\ \sigma_2)}{(\mathit{value\text{-}of}\ \ (\mathit{diff\text{-}exp}\ \ \mathit{exp}1\ \ \mathit{exp}2)\ \ \rho\ \ \sigma_0) = ((\mathit{num\text{-}val}\ \ \mathit{val}1{-}\mathit{val}2)\ \ \sigma_2)}$$

Language with Explicit References

Specification

- (value-of (const-exp n) ρ σ) = ((numval n), σ)
- $$\frac{(\textit{value-of}\ \ \textit{exp}1\ \ \rho\ \ \sigma_0)\ =\ (\textit{val}1,\ \sigma_1)\ \wedge\ (\textit{value-of}\ \ \textit{exp}2\ \ \rho\ \ \sigma_1)\ =\ (\textit{val}2,\ \sigma_2)}{(\textit{value-of}\ \ (\textit{diff-exp}\ \ \textit{exp}1\ \ \textit{exp}2)\ \ \rho\ \ \sigma_0)\ =\ ((\textit{num-val}\ \ \textit{val}1 - \textit{val}2)\ \ \sigma_2)}$$
- $$\frac{(\textit{value-of} e1\ \ \rho\ \ \sigma_0)\ =\ (v1, \sigma_1)}{(\textit{value-of}(\textit{if-exp}\ \ e1\ \ e2\ \ e3)\ \ \rho\ \ \sigma_0) = \begin{cases} ((\textit{value-of}\ \ e2\ \ \rho\ \ \sigma_1), \sigma_2) & \text{if (exp}\rightarrow\text{val v1)} = \#\text{t} \\ ((\textit{value-of}\ \ e3\ \ \rho\ \ \sigma_1), \sigma_2) & \text{otherwise} \end{cases}}$$

Language with Explicit References

Specification

- $\dfrac{(\textit{value-of}\ \ \textit{exp}\ \ \rho\ \ \sigma_0)\ =\ (\textit{val}1,\ \sigma_1)\ \ l \notin \textit{dom}(\sigma_0)}{(\textit{value-of}\ \ (\textit{newref-exp}\ \ \textit{exp})\ \ \rho\ \ \sigma_0) = ((\textit{ref-val}\ \ l),\ [l = \textit{val}1]\sigma_1)}$
- l is a new store location

Language with Explicit References

Specification

- $\dfrac{(\textit{value-of}\ \ \textit{exp}\ \ \rho\ \ \sigma_0)\ =\ (\textit{val}1,\ \sigma_1)\ \ l \notin \textit{dom}(\sigma_0)}{(\textit{value-of}\ \ (\textit{newref-exp}\ \ \textit{exp})\ \ \rho\ \ \sigma_0)=((\textit{ref-val}\ \ l),\ [l=\textit{val}1]\sigma_1)}$
- l is a new store location
- $\dfrac{(\textit{value-ofexp}\ \ \rho\ \ \sigma_0)\ =\ (l,\ \sigma_1)}{(\textit{value-of}\ \ (\textit{deref-exp}\ \ \textit{exp})\ \ \rho\ \ \sigma_0)\ =\ (\sigma_1(l),\ \sigma_1)}$

Language with Explicit References

Specification

- $\dfrac{(\textit{value-of}\ \ \textit{exp}\ \ \rho\ \ \sigma_0)\ =\ (\textit{val}1,\ \sigma_1)\ \ l \notin \textit{dom}(\sigma_0)}{(\textit{value-of}\ \ (\textit{newref-exp}\ \ \textit{exp})\ \ \rho\ \ \sigma_0)=((\textit{ref-val}\ \ l),\ [l=\textit{val}1]\sigma_1)}$
- l is a new store location
- $\dfrac{(\textit{value-ofexp}\ \ \rho\ \ \sigma_0)\ =\ (l,\ \sigma_1)}{(\textit{value-of}\ \ (\textit{deref-exp}\ \ \textit{exp})\ \ \rho\ \ \sigma_0)\ =\ (\sigma_1(l),\ \sigma_1)}$
- $\dfrac{(\textit{value-of}\ \ \textit{exp}1\ \ \rho\ \ \sigma_0)\ =\ (l,\ \sigma_1)\ \wedge\ (\textit{value-of}\ \ \textit{exp}2\ \ \rho\ \ \sigma_1)=(\textit{val},\ \sigma_2)}{(\textit{value-of}(\textit{setref-exp}\ \ \textit{exp}1\ \ \textit{exp}2)\ \ \rho\ \ \sigma_0)\ =\ (\varnothing,\ [l=\textit{val}]\sigma_2)}$

Language with Explicit References

Implementation

- Grammar

```
(expression ("begin" expression (arbno ";" expression) "end")
            begin-exp)

(expression ("newref" "(" expression ")") newref-exp)

(expression ("deref" "(" expression ")") deref-exp)

(expression ("setref" "(" expression "," expression ")")
            setref-exp)
```

Language with Explicit References

Implementation

- Design choice: the store is a global variable
- Design choice: Represent the store as a (listof expval)

Language with Explicit References

Implementation

- Design choice: the store is a global variable
- Design choice: Represent the store as a (listof expval)
-
```
;; reference? : RacketVal --> Bool
(define (reference? v) (and (integer? v) (>= v 0)))
```

Language with Explicit References

Implementation

- Design choice: the store is a global variable
- Design choice: Represent the store as a (listof expval)
- ```
  ;; reference? : RacketVal --> Bool
  (define (reference? v) (and (integer? v) (>= v 0)))
  ```
- ```
  ;; the-store: the current state of the  store
  (define the-store 'uninitialized)
  ```

Language with Explicit References

Implementation

- Design choice: the store is a global variable
- Design choice: Represent the store as a (listof expval)
- ```
  ;; reference? : RacketVal --> Bool
  (define (reference? v) (and (integer? v) (>= v 0)))
  ```
- ```
  ;; the-store: the current state of the  store
  (define the-store 'uninitialized)
  ```
- ```
  ;; empty-store :  --> store
  (define (empty-store) '())
  ```
  ```

Language with Explicit References

Implementation

- Design choice: the store is a global variable
- Design choice: Represent the store as a (listof expval)
- ```
  ;; reference? : RacketVal --> Bool
  (define (reference? v) (and (integer? v) (>= v 0)))
  ```
- ```
  ;; the-store: the current state of the  store
  (define the-store 'uninitialized)
  ```
- ```
  ;; empty-store :  --> store
  (define (empty-store) '())
  ```
- ```
  ;; initialize-store! :  --> store
  (define (initialize-store!) (set! the-store (empty-store)))
  ```

Language with Explicit References

Implementation

- Design choice: the store is a global variable
- Design choice: Represent the store as a (listof expval)
- ```
  ;; reference? : RacketVal --> Bool
  (define (reference? v) (and (integer? v) (>= v 0)))
  ```
- ```
  ;; the-store: the current state of the  store
  (define the-store 'uninitialized)
  ```
- ```
  ;; empty-store :  --> store
  (define (empty-store) '())
  ```
- ```
  ;; initialize-store! :  --> store
  (define (initialize-store!) (set! the-store (empty-store)))
  ```
- ```
  ;; newref : expval --> ref
  (define (newref val)
    (let ((next-ref (length the-store)))
      (set! the-store (append the-store (list val)))
      next-ref))
  ```
  ```

Language with Explicit References

Implementation

- Design choice: the store is a global variable
- Design choice: Represent the store as a (listof expval)
- ```
  ;; reference? : RacketVal --> Bool
  (define (reference? v) (and (integer? v) (>= v 0)))
  ```
- ```
  ;; the-store: the current state of the  store
  (define the-store 'uninitialized)
  ```
- ```
  ;; empty-store :  --> store
  (define (empty-store) '())
  ```
- ```
  ;; initialize-store! :  --> store
  (define (initialize-store!) (set! the-store (empty-store)))
  ```
- ```
  ;; newref : expval --> ref
  (define (newref val)
    (let ((next-ref (length the-store)))
      (set! the-store (append the-store (list val)))
      next-ref))
  ```
- ```
  ;; deref : ref --> expval
  (define (deref ref) (list-ref the-store ref))
  ```

Language with Explicit References

Implementation

- Design choice: the store is a global variable
- Design choice: Represent the store as a (listof expval)
- ```
  ;; reference? : RacketVal --> Bool
  (define (reference? v) (and (integer? v) (>= v 0)))
  ```
- ```
  ;; the-store: the current state of the  store
  (define the-store 'uninitialized)
  ```
- ```
  ;; empty-store :  --> store
  (define (empty-store) '())
  ```
- ```
  ;; initialize-store! :  --> store
  (define (initialize-store!) (set! the-store (empty-store)))
  ```
- ```
  ;; newref : expval --> ref
  (define (newref val)
    (let ((next-ref (length the-store)))
      (set! the-store (append the-store (list val)))
      next-ref))
  ```
- ```
  ;; deref : ref --> expval
  (define (deref ref) (list-ref the-store ref))
  ```
- ```
  ;; setref : ref expval --> expval
  (define (setref! ref new-expval)
    (set! the-store (append (take the-store ref)
                            (list new-expval)
                            (drop the-store (add1 ref)))))
  ```
  ```

Language with Explicit References

Implementation

- ```
  ;; value-of-program : program --> expval
  (define (value-of-program pgm)
    (begin
      (initialize-store!)
      (cases program pgm
        (a-program (exp1)
                   (value-of exp1 (empty-env))))))
  ```
  ```

Language with Explicit References

Implementation

- ```
  (define (value-of exp env)
    (cases expression exp
      (const-exp (num) (num-val num))
      (true-exp () (bool-val #t))
      (false-exp () (bool-val #f))
  ```
  ```

Language with Explicit References

Implementation

- ```
  (define (value-of exp env)
    (cases expression exp
      (const-exp (num) (num-val num))
      (true-exp () (bool-val #t))
      (false-exp () (bool-val #f))
  ```
- ```
      (var-exp (var) (apply-env env var))
  ```

Language with Explicit References

Implementation

- ```
  (define (value-of exp env)
    (cases expression exp
      (const-exp (num) (num-val num))
      (true-exp () (bool-val #t))
      (false-exp () (bool-val #f))
  ```
- ```
      (var-exp (var) (apply-env env var))
  ```
- ```
      (diff-exp (exp1 exp2)
                (let ((num1 (expval2num (value-of exp1 env)))
                      (num2 (expval2num (value-of exp2 env))))
                  (num-val (- num1 num2))))
  ```
  ```

Language with Explicit References

Implementation

- ```
  (define (value-of exp env)
    (cases expression exp
      (const-exp (num) (num-val num))
      (true-exp () (bool-val #t))
      (false-exp () (bool-val #f))
  ```
- ```
      (var-exp (var) (apply-env env var))
  ```
- ```
      (diff-exp (exp1 exp2)
                (let ((num1 (expval2num (value-of exp1 env)))
                      (num2 (expval2num (value-of exp2 env))))
                  (num-val (- num1 num2))))
  ```
- ```
      (zero?-exp (exp1)
                 (let ((val1 (expval2num (value-of exp1 env))))
                   (if (zero? val1)
                       (bool-val #t)
                       (bool-val #f))))
  ```

Language with Explicit References

Implementation

- ```
  (define (value-of exp env)
    (cases expression exp
      (const-exp (num) (num-val num))
      (true-exp () (bool-val #t))
      (false-exp () (bool-val #f))
  ```
- ```
      (var-exp (var) (apply-env env var))
  ```
- ```
      (diff-exp (exp1 exp2)
                (let ((num1 (expval2num (value-of exp1 env)))
                      (num2 (expval2num (value-of exp2 env))))
                  (num-val (- num1 num2))))
  ```
- ```
      (zero?-exp (exp1)
                 (let ((val1 (expval2num (value-of exp1 env))))
                   (if (zero? val1)
                       (bool-val #t)
                       (bool-val #f))))
  ```
- ```
      (if-exp (exp1 exp2 exp3)
              (let ((val1 (value-of exp1 env)))
                (if (expval2bool val1)
                    (value-of exp2 env)
                    (value-of exp3 env))))
  ```
  ```

Language with Explicit References

Implementation

- ```
  (define (value-of exp env)
    (cases expression exp
      (const-exp (num) (num-val num))
      (true-exp () (bool-val #t))
      (false-exp () (bool-val #f))
  ```
- ```
      (var-exp (var) (apply-env env var))
  ```
- ```
      (diff-exp (exp1 exp2)
                (let ((num1 (expval2num (value-of exp1 env)))
                      (num2 (expval2num (value-of exp2 env))))
                  (num-val (- num1 num2))))
  ```
- ```
      (zero?-exp (exp1)
                 (let ((val1 (expval2num (value-of exp1 env))))
                   (if (zero? val1)
                       (bool-val #t)
                       (bool-val #f))))
  ```
- ```
      (if-exp (exp1 exp2 exp3)
              (let ((val1 (value-of exp1 env)))
                (if (expval2bool val1)
                    (value-of exp2 env)
                    (value-of exp3 env))))
  ```
- ```
      (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))))

      .
      .
      .
  ```

Language with Explicit References

Implementation

- ```
  (define (value-of exp env)
    (cases expression exp
      .
      .
      .
      (proc-exp (params body)
                (proc-val (procedure params body (vector env))))
  ```
  ```

Language with Explicit References

Implementation

- ```
  (define (value-of exp env)
    (cases expression exp
        .
        .
        .
        (proc-exp (params body)
                  (proc-val (procedure params body (vector env))))
  ```
- ```
        (call-exp (rator rands)
                  (let [(proc (expval2proc (value-of rator env)))
                        (args (map (lambda (rand) (value-of rand env)) rands))]
                    (apply-procedure proc args)))
  ```

Language with Explicit References

Implementation

- ```
  (define (value-of exp env)
    (cases expression exp
      .
      .
      .
      (proc-exp (params body)
                (proc-val (procedure params body (vector env))))
  ```
- ```
      (call-exp (rator rands)
                (let [(proc (expval2proc (value-of rator env)))
                      (args (map (lambda (rand) (value-of rand env)) rands))]
                  (apply-procedure proc args)))
  ```
- ```
      (letrec-exp (names params bodies letrec-body)
                  (value-of letrec-body (mk-letrec-env names params bodies env)))
  ```
  ```

Language with Explicit References

Implementation

- ```
  (define (value-of exp env)
    (cases expression exp
        .
        .
        .
        (proc-exp (params body)
                  (proc-val (procedure params body (vector env))))
  ```
- ```
        (call-exp (rator rands)
                  (let [(proc (expval2proc (value-of rator env)))
                        (args (map (lambda (rand) (value-of rand env)) rands))]
                    (apply-procedure proc args)))
  ```
- ```
        (letrec-exp (names params bodies letrec-body)
                    (value-of letrec-body (mk-letrec-env names params bodies env)))
  ```
- ```
        (begin-exp (exp exps)
                   (foldl (lambda (e v) (value-of e env)) (value-of exp env) exps))
  ```

Language with Explicit References

Implementation

- ```
  (define (value-of exp env)
    (cases expression exp
      .
      .
      .
      (proc-exp (params body)
                (proc-val (procedure params body (vector env))))
  ```
- ```
      (call-exp (rator rands)
                (let [(proc (expval2proc (value-of rator env)))
                      (args (map (lambda (rand) (value-of rand env)) rands))]
                  (apply-procedure proc args)))
  ```
- ```
      (letrec-exp (names params bodies letrec-body)
                  (value-of letrec-body (mk-letrec-env names params bodies env)))
  ```
- ```
      (begin-exp (exp exps)
                 (foldl (lambda (e v) (value-of e env)) (value-of exp env) exps))
  ```
- ```
      (newref-exp (exp1)
                  (let ((v1 (value-of exp1 env)))
                    (ref-val (newref v1))))
  ```
  ```

Language with Explicit References

Implementation

- ```
  (define (value-of exp env)
    (cases expression exp
      .
      .
      .
      (proc-exp (params body)
                (proc-val (procedure params body (vector env))))
  ```
- ```
      (call-exp (rator rands)
                (let [(proc (expval2proc (value-of rator env)))
                      (args (map (lambda (rand) (value-of rand env)) rands))]
                  (apply-procedure proc args)))
  ```
- ```
      (letrec-exp (names params bodies letrec-body)
                  (value-of letrec-body (mk-letrec-env names params bodies env)))
  ```
- ```
      (begin-exp (exp exps)
                 (foldl (lambda (e v) (value-of e env)) (value-of exp env) exps))
  ```
- ```
      (newref-exp (exp1)
                  (let ((v1 (value-of exp1 env)))
                    (ref-val (newref v1))))
  ```
- ```
      (deref-exp (exp1)
                 (let ((v1 (value-of exp1 env)))
                   (let ((ref1 (expval2ref v1)))
                     (deref ref1))))
  ```

Language with Explicit References

Implementation

- ```
  (define (value-of exp env)
    (cases expression exp
      .
      .
      .
      (proc-exp (params body)
                (proc-val (procedure params body (vector env))))
  ```
- ```
      (call-exp (rator rands)
                (let [(proc (expval2proc (value-of rator env)))
                      (args (map (lambda (rand) (value-of rand env)) rands))]
                  (apply-procedure proc args)))
  ```
- ```
      (letrec-exp (names params bodies letrec-body)
                  (value-of letrec-body (mk-letrec-env names params bodies env)))
  ```
- ```
      (begin-exp (exp exps)
                 (foldl (lambda (e v) (value-of e env)) (value-of exp env) exps))
  ```
- ```
      (newref-exp (exp1)
                  (let ((v1 (value-of exp1 env)))
                    (ref-val (newref v1))))
  ```
- ```
      (deref-exp (exp1)
                 (let ((v1 (value-of exp1 env)))
                   (let ((ref1 (expval2ref v1)))
                     (deref ref1))))
  ```
- ```
      (setref-exp (exp1 exp2)
                  (let ((ref (expval2ref (value-of exp1 env))))
                    (let ((v2 (value-of exp2 env)))
                      (begin
                        (setref! ref v2)
                        (num-val -1))))))
  ```
  ```

Language with Explicit References

Homework

- 4.1, 4.2, 4.4, 4.8, 4.9

Language with Implicit References

- Most modern PLs package common patterns of allocation, dereferencing, and mutation
- Programmers do not need to worry about these operations

Language with Implicit References

- Most modern PLs package common patterns of allocation, dereferencing, and mutation
- Programmers do not need to worry about these operations
- Every variable denotes a reference
- References are no longer expressed values and exist only as bindings of vars

```
expval = int + bool + proc
denval = ref(expval)
```

Language with Implicit References

- Most modern PLs package common patterns of allocation, dereferencing, and mutation
- Programmers do not need to worry about these operations
- Every variable denotes a reference
- References are no longer expressed values and exist only as bindings of vars

  ```
  expval = int + bool + proc
  denval = ref(expval)
  ```

- Locations are created with each binding operation: procedure call, let, and letrec

Language with Implicit References

- Most modern PLs package common patterns of allocation, dereferencing, and mutation
- Programmers do not need to worry about these operations
- Every variable denotes a reference
- References are no longer expressed values and exist only as bindings of vars

  ```
  expval = int + bool + proc
  denval = ref(expval)
  ```
- Locations are created with each binding operation: procedure call, let, and letrec
- What happens when the interpreter encounters a var-exp?
  - env look-up to find the location to which it's bound
  - look-up in the store to find the value at that location
  - two-level system for var-exps

State

- The content of a location can be changed (or mutated)
- expression $\rightarrow$ set identifier $=$ expression
- the identifier is *not* an expression; not evaluated
- vars are mutable

State

- The content of a location can be changed (or mutated)
- expression $\rightarrow$ set identifier $=$ expression
- the identifier is *not* an expression; not evaluated
- vars are mutable
- Extend LETREC language and implement call-by-value semantics
- *Values* are passed to every function
- Formal parameters bound to locations of operand values
- It is the most common form of parameter passing

State

- The content of a location can be changed (or mutated)
- expression $\rightarrow$ set identifier $=$ expression
- the identifier is *not* an expression; not evaluated
- vars are mutable
- Extend LETREC language and implement call-by-value semantics
- *Values* are passed to every function
- Formal parameters bound to locations of operand values
- It is the most common form of parameter passing
- Why are chains of references not possible?

State

- The content of a location can be changed (or mutated)
- expression $\rightarrow$ set identifier $=$ expression
- the identifier is *not* an expression; not evaluated
- vars are mutable
- Extend LETREC language and implement call-by-value semantics
- *Values* are passed to every function
- Formal parameters bound to locations of operand values
- It is the most common form of parameter passing
- Why are chains of references not possible?
- Refs are not expressed values

State

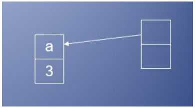
- Consider

```
let a = 3
in let p = proc (x) set x = 4
   in begin (p a); a end
```

State

- Consider

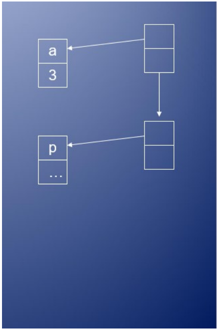
```
let a = 3
in let p = proc (x) set x = 4
   in begin (p a); a end
```

- 

State

- Consider

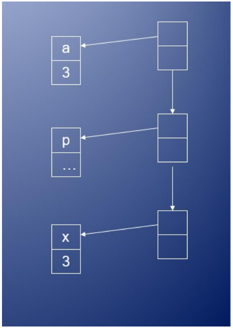
```
let a = 3
in let p = proc (x) set x = 4
   in begin (p a); a end
```

-


State

- Consider

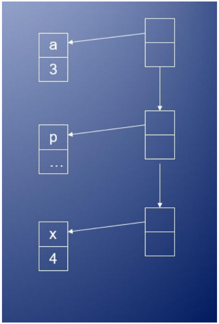
```
let a = 3
in let p = proc (x) set x = 4
   in begin (p a); a end
```

- 

State

- Consider

```
let a = 3
in let p = proc (x) set x = 4
   in begin (p a); a end
```

- 

- Returns 3

State

Specification

- $(\textit{value-of}\,(\textit{var-expv})\ \rho\ \sigma)\ =\ (\sigma(\rho(v)),\ \sigma)$
- Get v's binding (a reference) and access store for v's expval
- The store is unchanged

State

Specification

- $(\textit{value-of}\,(\textit{var-expv})\ \rho\ \sigma)\ =\ (\sigma(\rho(v)),\ \sigma)$
- Get v's binding (a reference) and access store for v's expval
- The store is unchanged
- $\dfrac{(\textit{value-of}\ \ \textit{exp}1\ \ \rho\ \ \sigma_0)\ =\ (\textit{val}1,\ \sigma_1)}{(\textit{value-of}\,(\textit{set-exp}\ \ v\ \ \textit{exp}1)\ \ \rho\ \ \sigma_0)\ =\ (\varnothing,\ [\sigma(v){=}\textit{val}1]\sigma_1)}$
- The location of v is changed to store val1
- The original value stored in σ(v) is lost forever

State

Specification

- $(\textit{value-of}\,(\textit{var-exp}\,v)\ \rho\ \sigma)\ =\ (\sigma(\rho(v)),\ \sigma)$
- Get v's binding (a reference) and access store for v's expval
- The store is unchanged

- $\dfrac{(\textit{value-of}\ \ \textit{exp}1\ \ \rho\ \ \sigma_0)\ =\ (\textit{val}1,\ \sigma_1)}{(\textit{value-of}\,(\textit{set-exp}\ \ v\ \ \textit{exp}1)\ \ \rho\ \ \sigma_0)\ =\ (\varnothing,\ [\sigma(v){=}\textit{val}1]\sigma_1)}$
- The location of v is changed to store val1
- The original value stored in σ(v) is lost forever

- $(\textit{apply-procedure}\ (\textit{procedure}\ \ v\ \ b\ \ \rho)\ \textit{val}\ \sigma)$
  $=\ \ (\textit{value-of}\ \ b\ [v = l]\rho\ [l = \textit{val}]\sigma)$
- The body is evaluated in a store where l contains the value of the parameter and an environment that binds the parameter to l

State

Specification

- $(\mathit{value\text{-}of}\,(\mathit{var\text{-}exp}\,v)\ \rho\ \sigma)\ =\ (\sigma(\rho(v)),\ \sigma)$
- Get v's binding (a reference) and access store for v's expval
- The store is unchanged

- $\dfrac{(\mathit{value\text{-}of}\ \ \mathit{exp1}\ \ \rho\ \ \sigma_0)\ =\ (\mathit{val1},\ \sigma_1)}{(\mathit{value\text{-}of}\,(\mathit{set\text{-}exp}\ \ v\ \ \mathit{exp1})\ \ \rho\ \ \sigma_0)\ =\ (\varnothing,\ [\sigma(v){=}\mathit{val1}]\sigma_1)}$
- The location of v is changed to store val1
- The original value stored in σ(v) is lost forever

- $(\mathit{apply\text{-}procedure}\ (\mathit{procedure}\ \ v\ \ b\ \ \rho)\ \mathit{val}\ \sigma)$
  $=\ (\mathit{value\text{-}of}\ \ b\ [v = l]\rho\ [l = \mathit{val}]\sigma)$
- The body is evaluated in a store where l contains the value of the parameter and an environment that binds the parameter to l
- $\dfrac{(\mathit{value\text{-}of}\ \ \mathit{exp1}\ \ \rho\ \ \sigma){=}(\mathit{val},\ \sigma_1)}{(\mathit{value\text{-}of}\ \ (\mathit{let\text{-}exp}\ \ \mathit{var}\ \ \mathit{exp1}\ \ \mathit{exp2})\ \ \rho\ \ \sigma){=}(\mathit{value\text{-}of}\ \ \mathit{exp2}\ \ [\mathit{var}{=}l]\rho\ \ [l{=}\mathit{val}]\sigma_1)}$
- Evaluate the body of the let-exp in a store where l contains the value of the local variable and the local variable is bound to l

State

- $$\frac{(\textit{value-of}\ \ e0\ \ \rho\ \ \sigma_0)=(p,\ \sigma_1)\ \wedge\ (\textit{value-of}\ \ e1\ \ \rho\ \ \sigma_1)=(v1,\sigma_2)\ \wedge\ (\textit{value-of}\ \ e2\ \ \rho\ \ \sigma_2)=(v2,\ \sigma_3)\wedge\ \ldots}{(\textit{value-of}\ \ (\textit{call-exp}\ \ e0\ \ e1\ldots en)\ \ \rho\ \ \sigma_0)=(\textit{apply-procedure}\ \ p\ \ v1\ldots vn\ \ \sigma_{n+1})}$$
- Evaluate all expressions using the given environment
- Evaluate *ei* using σ_i
- Apply the proc to the args using the store state after evaluating all expressions

State

- $$\frac{(\textit{value-of}\ \ e0\ \ \rho\ \ \sigma_0)=(p,\ \sigma_1)\ \wedge\ (\textit{value-of}\ \ e1\ \ \rho\ \ \sigma_1)=(v1,\sigma_2)\ \wedge\ (\textit{value-of}\ \ e2\ \ \rho\ \ \sigma_2)=(v2,\ \sigma_3)\wedge\ \ldots}{(\textit{value-of}\ \ (\textit{call-exp}\ \ e0\ \ e1\ldots en)\ \ \rho\ \ \sigma_0)=(\textit{apply-procedure}\ \ p\ \ v1\ldots vn\ \ \sigma_{n+1})}$$
- Evaluate all expressions using the given environment
- Evaluate *ei* using σ_i
- Apply the proc to the args using the store state after evaluating all expressions

- $$\frac{\rho_n=[n_1=l_1\ldots n_n=l_n]\rho\ \wedge\ p1=(\textit{proc-val}\ \ n_1\ \ p_1\ \ e_1\ \ \rho_n)\ \wedge\ldots\wedge\ pn=(\textit{proc-val}\ \ n_n\ \ p_n\ \ e_n\ \ \rho_n)}{(\textit{v-o}\ \ (\textit{letrec-exp}\ \ n_1\ldots n_n\ \ p_1\ldots p_n\ \ e_1\ldots e_n\ \ e_{n+1})\ \ \rho\ \ \sigma)=(\textit{v-o}\ \ e_{n+1}\ \ \rho_n\ \ [l_1=p1\ldots l_n=pn]\sigma)}$$
- v-o = value-of
- All procs are allocated in the store

State

Implementation

- ```
  (expression
       ("begin" expression (arbno ";" expression) "end")
       begin-exp)

  (expression ("set" identifier "=" expression)  set-exp)
  ```
  ```

State

Implementation

- ```
  (expression
       ("begin" expression (arbno ";" expression) "end")
       begin-exp)

  (expression ("set" identifier "=" expression)  set-exp)
  ```
- The store is the same as with Explicit Refs
  ```

State

Implementation

- ```
  (expression
       ("begin" expression (arbno ";" expression) "end")
       begin-exp)

  (expression ("set" identifier "=" expression)  set-exp)
  ```
- The store is the same as with Explicit Refs
- ```
  (define-datatype expval expval?
    (num-val
     (value number?))
    (bool-val
     (boolean boolean?))
    (proc-val
     (proc proc?)))
  ```
- Unlike Explicit Refs, no ref-val

State

Implementation

- ```
  (expression
       ("begin" expression (arbno ";" expression) "end")
       begin-exp)

  (expression ("set" identifier "=" expression)  set-exp)
  ```
- The store is the same as with Explicit Refs
- ```
  (define-datatype expval expval?
    (num-val
     (value number?))
    (bool-val
     (boolean boolean?))
    (proc-val
     (proc proc?)))
  ```
- Unlike Explicit Refs, no ref-val
- Same as Explicit Refs

  ```
  (define (value-of-program pgm)
    (begin
      (initialize-store!)
      (cases program pgm
        (a-program (exp1)
                   (value-of exp1 (empty-env))))))
  ```

State

Implementation

- ```
  (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))
  ```
  ```

State

Implementation

- ```
  (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))
  ```
  ```

State

Implementation

- ```
  (check-equal? (eval "let a = 3
                       in let p = proc (x) set x = 4
                          in begin
                               (p a);
                               a
                             end")
                (num-val 3))
  ```

  ```
  (check-equal? (eval "let x = 0
                       in letrec f (x) = set x = +(x, 1)
                                 g (a) = set x = +(x, 2)
                          in begin
                               (f x);
                               (g x);
                               x
                             end")
                (num-val 2))
  ```
  ```

State

Implementation

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

    (const-exp (num) (num-val num))

    (true-exp () (bool-val #t))

    (false-exp () (bool-val #f))

    (var-exp (var) (deref (apply-env env var)))

    (diff-exp (exp1 exp2)
              (let ((num1 (expval2num (value-of exp1 env)))
                    (num2 (expval2num (value-of exp2 env))))
                (num-val (- num1 num2))))

    (zero?-exp (exp1)
               (let ((val1 (expval2num (value-of exp1 env))))
                 (if (zero? val1)
                     (bool-val #t)
                     (bool-val #f))))
```

State

Implementation

- ```
  (if-exp (exp1 exp2 exp3)
      (let ((val1 (value-of exp1 env)))
        (if (expval2bool val1)
            (value-of exp2 env)
            (value-of exp3 env))))

  (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 (newref val) acc))
                             env
                             vars
                             vals))))

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

  (call-exp (rator rands)
            (let [(proc (expval2proc (value-of rator env)))
                  (args (map (lambda (rand) (value-of rand env))
              (apply-procedure proc args)))
  ```
  ```

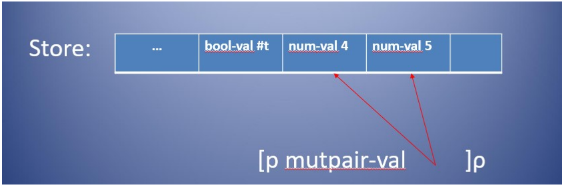
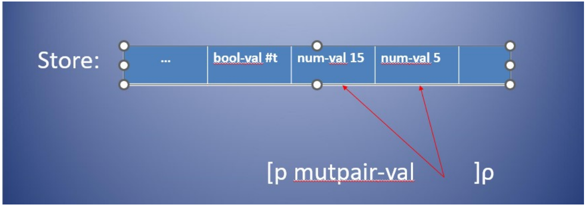
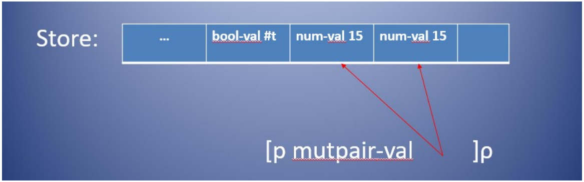
State

Implementation

- ```
  (letrec-exp (names params bodies rec-body)
    (value-of rec-body (mk-letrec-env names params bodies env)))

  (begin-exp (exp exps)
    (foldl (lambda (e v) (value-of e env))
           (value-of exp env)
           exps))

  (set-exp (var exp1)
           (begin
             (setref! (apply-env env var) (value-of exp1 env))
             (num-val 31))))
  ```
  ```

State

Implementation

- ```
  (define (mk-letrec-env names params bodies env)
    (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
                                         (newref 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))
        new-env)))
  ```
  ```

State

Implementation

- ```
  (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)
                                         (newref (cadr binding))
                                         acc))
                           saved-env
                           (map (lambda (p v) (list p v))
                                params
                                vals)))))))
  ```
  ```

- We will add mutable pairs to IMPLICIT-REFS

- We will add mutable pairs to IMPLICIT-REFS
- expval = int + bool + proc + mutpair
- mutpair = ref(expval) x ref(expval)

- We will add mutable pairs to IMPLICIT-REFS
- expval = int + bool + proc + mutpair
- mutpair = ref(expval) x ref(expval)
- DenVal = ref(expval)

- We will add mutable pairs to IMPLICIT-REFS
- expval = int + bool + proc + mutpair
- mutpair = ref(expval) x ref(expval)
- DenVal = ref(expval)
- Specification
  - → newpair: expval expval → mutpair
  - → left: mutpair → expval
  - → right: mutpair → expval
  - → setleft: mutpair expval → ∅
  - → setright: mutpair expval → ∅

- We will add mutable pairs to IMPLICIT-REFS
- expval = int + bool + proc + mutpair
- mutpair = ref(expval) x ref(expval)
- DenVal = ref(expval)
- Specification
  - $\rightarrow$ newpair: expval expval $\rightarrow$ mutpair
  - $\rightarrow$ left: mutpair $\rightarrow$ expval
  - $\rightarrow$ right: mutpair $\rightarrow$ expval
  - $\rightarrow$ setleft: mutpair expval $\rightarrow \emptyset$
  - $\rightarrow$ setright: mutpair expval $\rightarrow \emptyset$
-

```
(define-datatype expval expval?
  (num-val
   (value number?))
  (bool-val
   (boolean boolean?))
  (proc-val
   (proc proc?))
  (mutpair-val ;; new for mutable pairs
   (p mutpair?)))
```

- Grammar
  - → (expression ("newpair" "(" expression "," expression ")") newpair-exp)
  - → (expression ("left" "(" expression ")") left-exp)
  - → (expression ("setleft" expression "=" expression) setleft-exp)
  - → (expression ("right" "(" expression ")") right-exp)
  - → (expression ("setright" expression "=" expression) setright-exp)

- Let's trace

```
(eval "let p = newpair(4, 5)
       in begin
            setleft p = 15;
            setright p = 15;
            -(left(p), right(p))
          end")
```

- Let's trace

```
(eval "let p = newpair(4, 5)
       in begin
            setleft p = 15;
            setright p = 15;
            -(left(p), right(p))
          end")
```

- 

- Let's trace

```
(eval "let p = newpair(4, 5)
       in begin
            setleft p = 15;
            setright p = 15;
            -(left(p), right(p))
          end")
```

- 

- Let's trace

```
(eval "let p = newpair(4, 5)
       in begin
            setleft p = 15;
            setright p = 15;
            -(left(p), right(p))
          end")
```

  - 

- Returns (num-val 0)

- How can we represent a mutable pair?

- How can we represent a mutable pair?
- ```
  (define-datatype mutpair mutpair?
    (a-pair (left-loc reference?)
            (right-loc reference?)))
  ```
- Is this a good implementation choice?
  ```

- How can we represent a mutable pair?
- ```
  (define-datatype mutpair mutpair?
    (a-pair (left-loc reference?)
            (right-loc reference?)))
  ```
- Is this a good implementation choice?
- Does not take into account everything we know about mutable pairs
  - → The two locations are independently assignable
  - → Not independently allocated
  ```

- How can we represent a mutable pair?
- ```
  (define-datatype mutpair mutpair?
    (a-pair (left-loc reference?)
            (right-loc reference?)))
  ```
- Is this a good implementation choice?
- Does not take into account everything we know about mutable pairs
  - $\rightarrow$ The two locations are independently assignable
  - $\rightarrow$ Not independently allocated
- Consider newpair(4, 5) and σ

  σ = (...)
  σ = (... 4)
  σ = (... 4 5)
  ```

- How can we represent a mutable pair?
- ```
  (define-datatype mutpair mutpair?
    (a-pair (left-loc reference?)
            (right-loc reference?)))
  ```
- Is this a good implementation choice?
- Does not take into account everything we know about mutable pairs
  - $\rightarrow$ The two locations are independently assignable
  - $\rightarrow$ Not independently allocated
- Consider newpair(4, 5) and σ

  σ = (...)
  σ = (... 4)
  σ = (... 4 5)
- If the left is in position p in σ , where is the right?
  ```

- How can we represent a mutable pair?
-
```
(define-datatype mutpair mutpair?
  (a-pair (left-loc reference?)
          (right-loc reference?)))
```
- Is this a good implementation choice?
- Does not take into account everything we know about mutable pairs
  - $\rightarrow$ The two locations are independently assignable
  - $\rightarrow$ Not independently allocated
- Consider newpair(4, 5) and σ

  σ = (...)
  σ = (... 4)
  σ = (... 4 5)
- If the left is in position p in σ , where is the right?
- What does this tell you?

- How can we represent a mutable pair?
- ```
  (define-datatype mutpair mutpair?
    (a-pair (left-loc reference?)
            (right-loc reference?)))
  ```
- Is this a good implementation choice?
- Does not take into account everything we know about mutable pairs
  - → The two locations are independently assignable
  - → Not independently allocated
- Consider newpair(4, 5) and σ

  σ = (...)
  σ = (... 4)
  σ = (... 4 5)
- If the left is in position p in σ , where is the right?
- What does this tell you?
- We can implement mutable pairs using a single reference
  ```

- ```
  ;; expval --> reference throws error
  (define (expval->mutpair v)
    (cases expval v
      (mutpair-val (ref) ref)
      (else (expval-extractor-error 'mutable-pair v))))
  ```
  ```

- ```
  ;; expval --> reference throws error
  (define (expval->mutpair v)
    (cases expval v
      (mutpair-val (ref) ref)
      (else (expval-extractor-error 'mutable-pair v))))
  ```
- ```
  ;; mutpair? : X -> Boolean
  (define (mutpair? v) (reference? v))
  ```

- ```
  ;; expval --> reference throws error
  (define (expval->mutpair v)
    (cases expval v
      (mutpair-val (ref) ref)
      (else (expval-extractor-error 'mutable-pair v))))
  ```
- ```
  ;; mutpair? : X -> Boolean
  (define (mutpair? v) (reference? v))
  ```
- ```
  ;; make-pair : expval expval -> mutpair
  (define (make-pair val1 val2)
    (let ((ref1 (newref val1)))
      (let ((ref2 (newref val2)))
        ref1)))
  ```
  ```

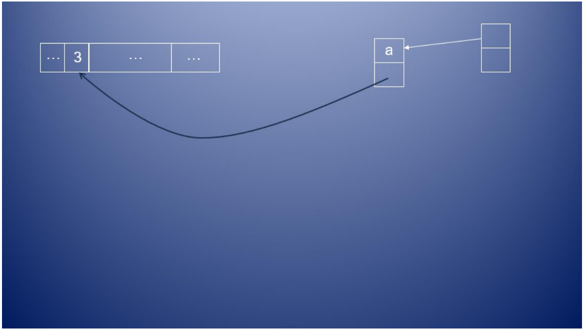
- ```
  ;; expval --> reference throws error
  (define (expval->mutpair v)
    (cases expval v
      (mutpair-val (ref) ref)
      (else (expval-extractor-error 'mutable-pair v))))
  ```
- ```
  ;; mutpair? : X -> Boolean
  (define (mutpair? v) (reference? v))
  ```
- ```
  ;; make-pair : expval expval -> mutpair
  (define (make-pair val1 val2)
    (let ((ref1 (newref val1)))
      (let ((ref2 (newref val2)))
       ref1)))
  ```
- ```
  ;; left : mutpair -> expval
  (define (left p) (deref p))

  ;; right : mutpair -> expval
  (define (right p) (deref (+ 1 p)))
  ```

- ```
  ;; expval --> reference throws error
  (define (expval->mutpair v)
    (cases expval v
      (mutpair-val (ref) ref)
      (else (expval-extractor-error 'mutable-pair v))))
  ```
- ```
  ;; mutpair? : X -> Boolean
  (define (mutpair? v) (reference? v))
  ```
- ```
  ;; make-pair : expval expval -> mutpair
  (define (make-pair val1 val2)
    (let ((ref1 (newref val1)))
      (let ((ref2 (newref val2)))
        ref1)))
  ```
- ```
  ;; left : mutpair -> expval
  (define (left p) (deref p))

  ;; right : mutpair -> expval
  (define (right p) (deref (+ 1 p)))
  ```
- ```
  ;; setleft : mutpair expval -> Unspecified
  (define (setleft p val) (setref! p val))

  ;; setright : mutpair expval -> Unspecified
  (define (setright p val) (setref! (+ 1 p) val))
  ```
  ```

-
```
(check-equal? (eval "let p = newpair(4, 5)
                     in left(p)")
              (num-val 4))

(check-equal? (eval "let p = newpair(4, 5)
                     in right(p)")
              (num-val 5))

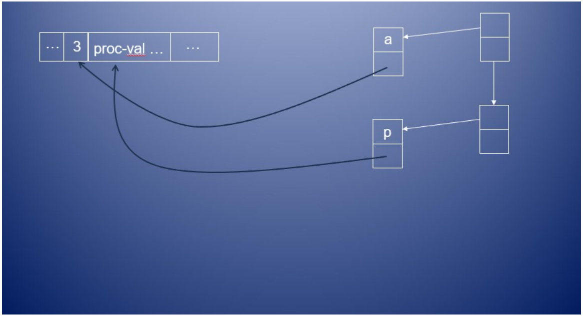
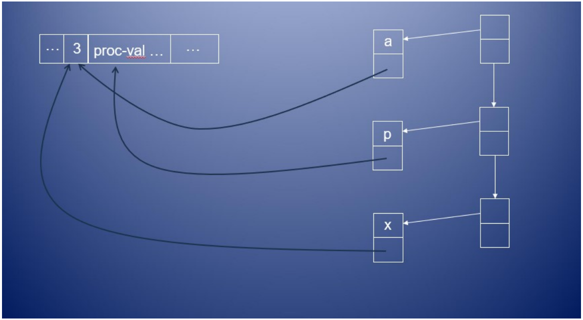
(check-equal? (eval "let p = newpair(4, 5)
                     in begin
                          setleft p = 15;
                          setright p = 15;
                          -(left(p), right(p))
                        end")
              (num-val 0))
```

- ```
  (define (value-of exp env)
    (cases expression exp
          ⋮
      (newpair-exp (exp1 exp2)
        (let ((v1 (value-of exp1 env))
              (v2 (value-of exp2 env)))
          (mutpair-val (make-pair v1 v2))))
  ```
  ```

- ```
  (define (value-of exp env)
    (cases expression exp
          ⋮
      (newpair-exp (exp1 exp2)
        (let ((v1 (value-of exp1 env))
              (v2 (value-of exp2 env)))
          (mutpair-val (make-pair v1 v2))))
  ```
- ```
      (left-exp (exp1)
        (let ((v1 (value-of exp1 env)))
          (let ((p1 (expval->mutpair v1)))
            (left p1))))
      (right-exp (exp1)
        (let ((v1 (value-of exp1 env)))
          (let ((p1 (expval->mutpair v1)))
            (right p1))))
  ```

- ```
  (define (value-of exp env)
    (cases expression exp
          ⋮
      (newpair-exp (exp1 exp2)
        (let ((v1 (value-of exp1 env))
              (v2 (value-of exp2 env)))
          (mutpair-val (make-pair v1 v2))))
  ```
- ```
      (left-exp (exp1)
        (let ((v1 (value-of exp1 env)))
          (let ((p1 (expval->mutpair v1)))
            (left p1))))
      (right-exp (exp1)
        (let ((v1 (value-of exp1 env)))
          (let ((p1 (expval->mutpair v1)))
            (right p1))))
  ```
- ```
      (setleft-exp (exp1 exp2)
        (let ((v1 (value-of exp1 env))
              (v2 (value-of exp2 env)))
          (let ((p (expval->mutpair v1)))
            (begin (setleft p v2)
                   (num-val 82))))) ;; this is a don't care value.
      (setright-exp (exp1 exp2)
        (let ((v1 (value-of exp1 env))
              (v2 (value-of exp2 env)))
          (let ((p (expval->mutpair v1)))
            (begin (setright p v2)
                   (num-val 83))))) ;; this is a don't care value.
  ```
  ```

- Homework: 4.28–4.30

Parameter Passing Variations

Call by Reference

- In call-by-value semantics the callee is isolated from the caller
- Assignments by the callee to its parameters can not be seen by the caller

Parameter Passing Variations

Call by Reference

- In call-by-value semantics the callee is isolated from the caller
- Assignments by the callee to its parameters can not be seen by the caller
- Sometimes it is desirable to pass in variables expecting the callee to make assigments to them
- This can be done by passing references to the callee instead of actual values
- This is known as *call-by-reference*

Parameter Passing Variations

Call by Reference

- In call-by-value semantics the callee is isolated from the caller
- Assignments by the callee to its parameters can not be seen by the caller
- Sometimes it is desirable to pass in variables expecting the callee to make assigments to them
- This can be done by passing references to the callee instead of actual values
- This is known as *call-by-reference*
- If an operand is a variable, then a reference to the variable's location is passed
- The formal parameter is bound to this location

Parameter Passing Variations

Call by Reference

- In call-by-value semantics the callee is isolated from the caller
- Assignments by the callee to its parameters can not be seen by the caller
- Sometimes it is desirable to pass in variables expecting the callee to make assigments to them
- This can be done by passing references to the callee instead of actual values
- This is known as *call-by-reference*
- If an operand is a variable, then a reference to the variable's location is passed
- The formal parameter is bound to this location
- If the operand is some other type of expression, then the formal parameter is bound to a new location containing the value of the operand
- Just like in call-by-value

Parameter Passing Variations

Call by Reference

- ```
  let a = 3
  p = proc (x) set x = 4
  in begin
      (p a);
      a
     end
  ```

- 
  ```

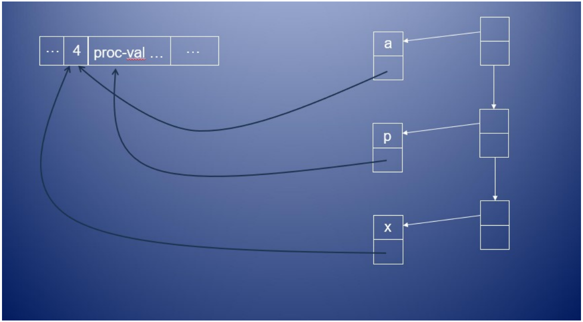
Parameter Passing Variations

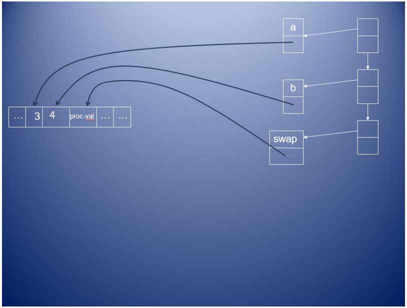
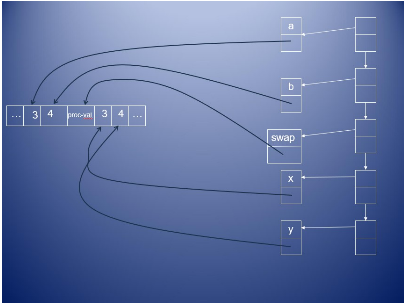
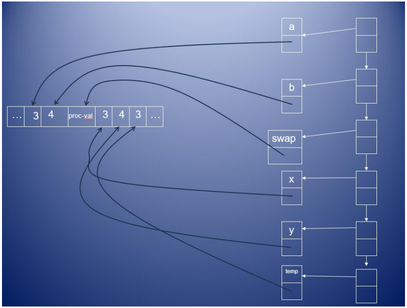
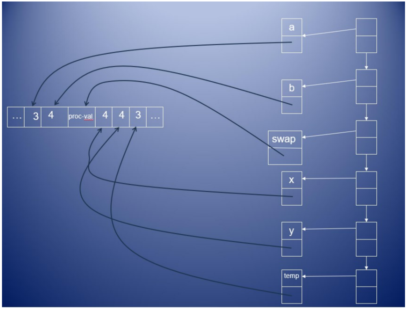
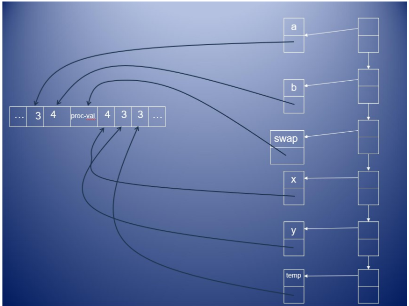
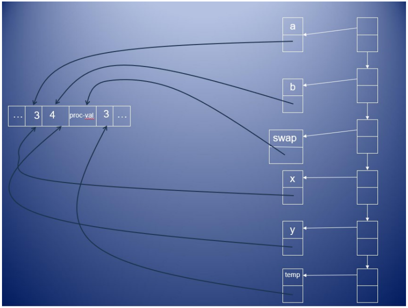
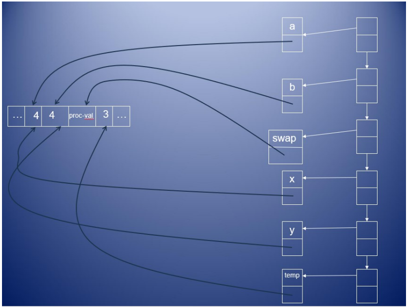
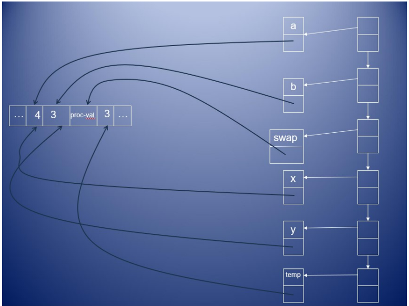
Call by Reference

- ```
  let a = 3
  p = proc (x) set x = 4
  in begin
      (p a);
      a
     end
  ```
- 
  ```

Parameter Passing Variations

Call by Reference

- ```
  let a = 3
  p = proc (x) set x = 4
  in begin
      (p a);
      a
     end
  ```
- 
  ```

Parameter Passing Variations

Call by Reference

- ```
  let a = 3
  p = proc (x) set x = 4
  in begin
      (p a);
      a
     end
  ```
  - 

- Returns 4
  ```

Parameter Passing Variations

Call by Reference

- Why use call-by-reference?
  - → Return multiple values (by making assignments to parameters)
  - → Implementation of common operations

Parameter Passing Variations

Call by Reference

- Call-by-Value

```
let a = 3
    b = 4
    swap = proc (x, y)
        let temp = x
        in begin
             set x = y
             set y = temp
           end
in begin
     swap(a b)
     -(a, b)
   end
```



Parameter Passing Variations

Call by Reference

- Call-by-Value

```
let a = 3
    b = 4
    swap = proc (x, y)
        let temp = x
        in begin
             set x = y
             set y = temp
           end
in begin
     swap(a b)
     -(a, b)
   end
```



Parameter Passing Variations

Call by Reference

- Call-by-Value

```
let a = 3
    b = 4
    swap = proc (x, y)
        let temp = x
        in begin
             set x = y
             set y = temp
           end
in begin
     swap(a b)
     -(a, b)
   end
```



Parameter Passing Variations

Call by Reference

- Call-by-Value

```
let a = 3
    b = 4
    swap = proc (x, y)
        let temp = x
        in begin
             set x = y
             set y = temp
           end
in begin
     swap(a b)
     -(a, b)
   end
```



Parameter Passing Variations

Call by Reference

- Call-by-Value

```
let a = 3
    b = 4
    swap = proc (x, y)
        let temp = x
        in begin
             set x = y
             set y = temp
           end
in begin
     swap(a b)
     -(a, b)
   end
```

Returns -1



Parameter Passing Variations

Call by Reference

- Call-by-Reference

```
let a = 3
    b = 4
    swap = proc (x, y)
        let temp = x
        in begin
             set x = y
             set y = temp
           end
in begin
     swap(a b)
     -(a, b)
   end
```



Parameter Passing Variations

Call by Reference

- Call-by-Reference

```
let a = 3
    b = 4
    swap = proc (x, y)
        let temp = x
        in begin
             set x = y
             set y = temp
           end
in begin
     swap(a b)
     -(a, b)
   end
```



Parameter Passing Variations

Call by Reference

- Call-by-Reference

```
let a = 3
    b = 4
    swap = proc (x, y)
        let temp = x
        in begin
             set x = y
             set y = temp
           end
in begin
     swap(a b)
     -(a, b)
   end
```

Returns 1



Parameter Passing Variations

Call by Reference

- Only change is for when new references are created:
  - $\rightarrow$ call-by-value: a new reference is created for every operand evaluated
  - $\rightarrow$ call-by-reference: a new reference is created for evaluation of an operand other than a variable
- Under call-by-reference we need a new location for some operands and not for others

Parameter Passing Variations

Call by Reference

- ```
  ;; apply-procedure : proc (listof expval) -> expval
  (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)
                                     (newref (cadr binding))
                                     acc))
                       saved-env
                       (map (lambda (p v) (list p v)) params vals)))))))
  ```
  Can't always allocate an argument in the store
  ```

Parameter Passing Variations

Call by Reference

- ```
  ;; apply-procedure : proc (listof expval) -> expval
  (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)
                                 (newref (cadr binding))
                                 acc))
                   saved-env
                   (map (lambda (p v) (list p v)) params vals)))))))
  ```
  Can't always allocate an argument in the store
- ```
  ;; apply-procedure : proc (listof ref) -> expval
  (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))))
  ```
  Decision made in the evaluation of a call-exp

Parameter Passing Variations

Call by Reference

- In value-of

```
(call-exp (rator rands)
  (let [(proc (expval2proc (value-of rator env)))
        (args (map (lambda (rand) (value-of rand env)) rands))]
    (apply-procedure proc args)))
```

`apply-procedure must be called with a (listof ref)`

Parameter Passing Variations

Call by Reference

- In value-of

```
(call-exp (rator rands)
  (let [(proc (expval2proc (value-of rator env)))
        (args (map (lambda (rand) (value-of rand env)) rands))]
    (apply-procedure proc args)))
```

apply-procedure must be called with a (listof ref)

- ```
  (call-exp (rator rands)
    (let [(proc (expval2proc (value-of rator env)))
          (args (map (lambda (rand) (value-of-rand rand env)) rands))
      (apply-procedure proc args)))
  ```

value-of-rand returns a reference
  ```

Parameter Passing Variations

Call by Reference

- In value-of

```
(call-exp (rator rands)
  (let [(proc (expval2proc (value-of rator env)))
        (args (map (lambda (rand) (value-of rand env)) rands))]
    (apply-procedure proc args)))
```

```
apply-procedure must be called with a (listof ref)
```

- ```
  (call-exp (rator rands)
    (let [(proc (expval2proc (value-of rator env)))
          (args (map (lambda (rand) (value-of-rand rand env)) rands))
      (apply-procedure proc args)))
  ```

```
value-of-rand returns a reference
```

- ```
  ;; value-of-rand : expression environment -> Ref
  ;; Purpose: For a var-exp return existing reference.
  ;;          Otherwise, return reference to a new cell.
  (define (value-of-rand exp env)
    (cases expression exp
      (var-exp (var) (apply-env env var))
      (else (newref (value-of exp env)))))
  ```

Parameter Passing Variations

Call by Reference

```
(check-equal? (eval "let a = 3
                     in let p = proc (x) set x = 4
                        in begin (p a); a end")
              (num-val 4))
(check-equal? (eval "let x = 0
                     in letrec f (x) = set x = +(x, 1)
                               g (a) = set x = +(x, 2)
                        in begin (f x);
                                 (g x);
                                 x
                           end")
              (num-val 3))
(check-equal?
 (eval "let swap = proc (a)
                    proc (b)
                      let t = a
                      in begin set a = b;  set b = t  end
        in let a = 33
           in let b = 44
              in begin ((swap a) b);
                       -(a, b)
                 end")
 (num-val 11))
```

Parameter Passing Variations

Lazy Evaluation: Call by Name

- Call-by-value and call-by-reference are eager
- Always find the value of each operand

Parameter Passing Variations

Lazy Evaluation: Call by Name

- Call-by-value and call-by-reference are eager
- Always find the value of each operand
- Lazy evaluation
- Operands not evaluated until needed
- Never needed $\rightarrow$ never evaluated

Parameter Passing Variations

Lazy Evaluation: Call by Name

- Call-by-value and call-by-reference are eager
- Always find the value of each operand
- Lazy evaluation
- Operands not evaluated until needed
- Never needed $\rightarrow$ never evaluated
- Is this useful?

Parameter Passing Variations

Lazy Evaluation: Call by Name

- ```
  letrec compute-ints-from-n (n) = (compute-ints-from-n +(n, 1))
  in let f = proc (k) 42
     in (f (compute-ints-from-n 100))
  ```
- What should this program return?
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Name

- ```
  letrec compute-ints-from-n (n) = (compute-ints-from-n +(n, 1))
  in let f = proc (k) 42
     in (f (compute-ints-from-n 100))
  ```
- What should this program return?
- It should return 42, but does not. Why?
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Name

- ```
  letrec compute-ints-from-n (n) = (compute-ints-from-n +(n, 1))
  in let f = proc (k) 42
     in (f (compute-ints-from-n 100))
  ```
- What should this program return?
- It should return 42, but does not. Why?
- Under lazy evaluation this program returns 42
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Name

- ```
  letrec compute-ints-from-n (n) = (compute-ints-from-n +(n, 1))
  in let f = proc (k) 42
     in (f (compute-ints-from-n 100))
  ```
- What should this program return?
- It should return 42, but does not. Why?
- Under lazy evaluation this program returns 42
- ```
  #lang eopl
  (require rackunit "../eopl-extras.rkt")

  (define (ints-from n) (stream-cons n (ints-from (+ n 1))))
  (define natnums (ints-from 0))
  (define (nth-natnum n) (stream-ref natnums n))
  (define (first-n-natnums n)
    (if (= n 0)
        (list (nth-natnum 0))
        (cons (nth-natnum n) (first-n-natnums (- n 1)))))

  (check-equal? (first-n-natnums 10)
                '(10 9 8 7 6 5 4 3 2 1 0))
  (check-equal? (first-n-natnums 15)
                '(15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0))
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Name

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

  ;; natnum --> natnum
  ;; Purpose: Return the kth Fibonacci number
  (define (fib k)
    (if (< k 2)
        1
        (+ (fib (- k 1)) (fib (- k 2)))))
  (define (the-fibs n) (stream-cons (fib n) (the-fibs (+ n 1))))
  (define fibs (the-fibs 0))
  (define (nth-fib n) (stream-ref fibs n))

  (check-equal?  (nth-fib 5)  8)
  (check-equal?  (nth-fib 10) 89)
  ```
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Name

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

  ;; natnum --> natnum
  ;; Purpose: Return the kth Fibonacci number
  (define (fib k)
    (if (< k 2)
        1
        (+ (fib (- k 1)) (fib (- k 2)))))
  (define (the-fibs n) (stream-cons (fib n) (the-fibs (+ n 1))))
  (define fibs (the-fibs 0))
  (define (nth-fib n) (stream-ref fibs n))

  (check-equal?  (nth-fib 5)  8)
  (check-equal?  (nth-fib 10) 89)
  ```
- ```
  (define the-doubles (stream-map (λ (n) (* 2 n)) natnums))

  (check-equal? (stream-ref the-doubles 10) 20)
  (check-equal? (stream-ref the-doubles 1287) 2574)
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Name

- An operand is not evaluated until needed

Parameter Passing Variations

Lazy Evaluation: Call by Name

- An operand is not evaluated until needed
- A bound var is associated with unevaluated expression (frozen)

Parameter Passing Variations

Lazy Evaluation: Call by Name

- An operand is not evaluated until needed
- A bound var is associated with unevaluated expression (frozen)
- When the value of the bound var is needed, then the expression is evaluated (thawed)

Parameter Passing Variations

Lazy Evaluation: Call by Name

- An operand is not evaluated until needed
- A bound var is associated with unevaluated expression (frozen)
- When the value of the bound var is needed, then the expression is evaluated (thawed)
- What does this require?

Parameter Passing Variations

Lazy Evaluation: Call by Name

- An operand is not evaluated until needed
- A bound var is associated with unevaluated expression (frozen)
- When the value of the bound var is needed, then the expression is evaluated (thawed)
- What does this require?
- The env that exists when the expr is frozen

Parameter Passing Variations

Lazy Evaluation: Call by Name

- An operand is not evaluated until needed
- A bound var is associated with unevaluated expression (frozen)
- When the value of the bound var is needed, then the expression is evaluated (thawed)
- What does this require?
- The env that exists when the expr is frozen
- ```
  (define-datatype thunk thunk?
    (a-thunk
      (exp1 expression?)
      (env environment?)))
  ```
- The expr in a thunk is evaluated when a proc needs the value of bound var
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Name

- Language
  - → let remains eager
  - → lazy evaluation of arguments
  - → effects

Parameter Passing Variations

Lazy Evaluation: Call by Name

- Language
  - $\rightarrow$ let remains eager
  - $\rightarrow$ lazy evaluation of arguments
  - $\rightarrow$ effects
- Values
  - $\rightarrow$ expval = int + bool + proc
  - $\rightarrow$ denval = ref(expval + thunk)

Parameter Passing Variations

Lazy Evaluation: Call by Name

- Language
  - $\rightarrow$ let remains eager
  - $\rightarrow$ lazy evaluation of arguments
  - $\rightarrow$ effects
- Values
  - $\rightarrow$ expval = int + bool + proc
  - $\rightarrow$ denval = ref(expval + thunk)
- New allocations policy
  - $\rightarrow$ var: pass its denotation (which is a reference; same as call-by-reference)
  - $\rightarrow$ not var: pass a ref to a new location storing a thunk for the unevaluated arg

Parameter Passing Variations

Lazy Evaluation: Call by Name

- ```
  ;; value-of-rand : expression environment -> Ref
  ;; Purpose: if the expression is a var-exp, then return the referen
  ;; otherwise,return a thunk for the given expression.
  (define (value-of-rand exp env)
    (cases expression exp
      (var-exp (var) (apply-env env var))
      (else
       (newref (a-thunk exp env))))) ← not a var-exp create thunk
  ```
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Name

- How do you evaluate a var-expr?

Parameter Passing Variations

Lazy Evaluation: Call by Name

- How do you evaluate a var-expr?
- $$\frac{w \ = \ \mathit{deref}(\rho(v))}{(\mathit{value\text{-}of}\ \ (\mathit{var\text{-}expv})\ \ \rho) = \text{if (expval? w) then w else (value-of-thunk w)}}$$

Parameter Passing Variations

Lazy Evaluation: Call by Name

- How do you evaluate a var-expr?
- $$\frac{w \;=\; \mathit{deref}(\rho(v))}{(\mathit{value\text{-}of}\;\;(\mathit{var\text{-}expv})\;\;\rho) \;=\; \text{if (expval? w) then w else (value-of-thunk w)}}$$
- change to value-of

```
(var-exp (var)
  (let ((ref1 (apply-env env var)))
    (let ((w (deref ref1)))
      (if (expval? w)
          w
  (value-of-thunk w)))))
```

Parameter Passing Variations

Lazy Evaluation: Call by Name

- Evaluating a thunk

```
;; value-of-thunk : thunk -> expval
;; Purpose: Evaluate the given thunk
(define (value-of-thunk th)
  (cases thunk th
    (a-thunk (exp1 saved-env)
             (value-of exp1 saved-env))))
```

Parameter Passing Variations

Lazy Evaluation: Call by Name

- Consider

```
let g = let counter = 10
            in proc (d) *(2, counter)
in (proc (x) +(x, x) (g 0))
```

Parameter Passing Variations

Lazy Evaluation: Call by Name

- Consider

```
let g = let counter = 10
            in proc (d) *(2, counter)
in (proc (x) +(x, x) (g 0))
```

- x is the thunk for (g 0)

Parameter Passing Variations

Lazy Evaluation: Call by Name

- Consider

```
let g = let counter = 10
            in proc (d) *(2, counter)
in (proc (x) +(x, x) (g 0))
```

- x is the thunk for (g 0)
- the first x forces the evaluation of the thunk → 20

Parameter Passing Variations

Lazy Evaluation: Call by Name

- Consider

```
let g = let counter = 10
            in proc (d) *(2, counter)
in (proc (x) +(x, x) (g 0))
```

- x is the thunk for (g 0)
- the first x forces the evaluation of the thunk → 20
- the second x forces the evaluation of the thunk → 20
- returns 40

Parameter Passing Variations

Lazy Evaluation: Call by Need

- Evaluating the same thunk seems wasteful

Parameter Passing Variations

Lazy Evaluation: Call by Need

- Evaluating the same thunk seems wasteful
- Solution: Evaluate it once and mutate it for its value

Parameter Passing Variations

Lazy Evaluation: Call by Need

- Evaluating the same thunk seems wasteful
- Solution: Evaluate it once and mutate it for its value
- Change in value-of

```
(var-exp (var)
  (let ((ref1 (apply-env env var)))
    (let ((w (deref ref1)))
      (if (expval? w)
          w
          (let ((v1 (value-of-thunk w)))
            (begin
              (setref! ref1 v1)
              v1))))))
```

Parameter Passing Variations

Lazy Evaluation: Call by Need

- Change how var-exp are evaluated

```
(var-exp (var)
  (let ((ref1 (apply-env env var)))
    (let ((w (deref ref1)))
      (if (expval? w)
          w
          (let ((v1 (value-of-thunk w)))
            (begin
              (setref! ref1 v1)
              v1)))))))
```

Parameter Passing Variations

Lazy Evaluation: Call by Need

- ```
  let g = let counter = 10
          in proc (d) *(2, counter)
  in (proc (x) +(x, x) (g 0))
  ```
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Need

- ```
  let g = let counter = 10
          in proc (d) *(2, counter)
  in (proc (x) +(x, x) (g 0))
  ```
- x is the thunk for (g 0)
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Need

- ```
  let g = let counter = 10
          in proc (d) *(2, counter)
  in (proc (x) +(x, x) (g 0))
  ```

- the first x forces the evaluation of the thunk to 20
- mutates x to 20
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Need

- ```
  let g = let counter = 10
          in proc (d) *(2, counter)
  in (proc (x) +(x, x) (g 0))
  ```

- the first x forces the evaluation of the thunk to 20
- mutates x to 20
- the second x (simply) returns its value of 20
- returns 40
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Need

- In the absence of side-effects, call-by-name and call-by-need always yield the same answer

Parameter Passing Variations

Lazy Evaluation: Call by Need

- In the absence of side-effects, call-by-name and call-by-need always yield the same answer
- In the presence of side-effects, it is easy to distinguish them

Parameter Passing Variations

Lazy Evaluation: Call by Need

- In the absence of side-effects, call-by-name and call-by-need always yield the same answer
- In the presence of side-effects, it is easy to distinguish them
- ```
  let g = let count = 0
          in proc (d)
           begin
                 set count = incr(count);
                 count
          end
  in (proc (x) +(x, x) (g 0) )
  ```
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Need

- In the absence of side-effects, call-by-name and call-by-need always yield the same answer
- In the presence of side-effects, it is easy to distinguish them
- ```
  let g = let count = 0
          in proc (d)
           begin
                 set count = incr(count);
                 count
          end
  in (proc (x) +(x, x) (g 0) )
  ```
- g returns the number of times it is called
- Thunk for (g 0) is passed as the argument to the function in the body of the let
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Need

- In the absence of side-effects, call-by-name and call-by-need always yield the same answer
- In the presence of side-effects, it is easy to distinguish them
- ```
  let g = let count = 0
          in proc (d)
           begin
                set count = incr(count);
                count
          end
  in (proc (x) +(x, x) (g 0) )
  ```
- g returns the number of times it is called
- Thunk for (g 0) is passed as the argument to the function in the body of the let
- call-by-name
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Need

- In the absence of side-effects, call-by-name and call-by-need always yield the same answer
- In the presence of side-effects, it is easy to distinguish them
- ```
  let g = let count = 0
          in proc (d)
           begin
                 set count = incr(count);
                 count
          end
  in (proc (x) +(x, x) (g 0) )
  ```
- g returns the number of times it is called
- Thunk for (g 0) is passed as the argument to the function in the body of the let
- call-by-name
- the first reference to x: sets count to 1 & returns 1 as the value of (g 0)
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Need

- In the absence of side-effects, call-by-name and call-by-need always yield the same answer
- In the presence of side-effects, it is easy to distinguish them
- ```
  let g = let count = 0
          in proc (d)
           begin
                 set count = incr(count);
                 count
          end
  in (proc (x) +(x, x) (g 0) )
  ```
- g returns the number of times it is called
- Thunk for (g 0) is passed as the argument to the function in the body of the let
- call-by-name
- the first reference to x: sets count to 1 & returns 1 as the value of (g 0)
- the second reference to x: sets count to 2 & returns 2 as the value of (g 0)
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Need

- In the absence of side-effects, call-by-name and call-by-need always yield the same answer
- In the presence of side-effects, it is easy to distinguish them
- ```
  let g = let count = 0
          in proc (d)
           begin
                 set count = incr(count);
                 count
          end
  in (proc (x) +(x, x) (g 0) )
  ```
- g returns the number of times it is called
- Thunk for (g 0) is passed as the argument to the function in the body of the let
- call-by-name
- the first reference to x: sets count to 1 & returns 1 as the value of (g 0)
- the second reference to x: sets count to 2 & returns 2 as the value of (g 0)
- $+(1, 2) = 3$
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Need

- In the absence of side-effects, call-by-name and call-by-need always yield the same answer
- In the presence of side-effects, it is easy to distinguish them
- ```
  let g = let count = 0
          in proc (d)
           begin
                 set count = incr(count);
                 count
          end
  in (proc (x) +(x, x) (g 0) )
  ```
- g returns the number of times it is called
- Thunk for (g 0) is passed as the argument to the function in the body of the let
- call-by-name
- the first reference to x: sets count to 1 & returns 1 as the value of (g 0)
- the second reference to x: sets count to 2 & returns 2 as the value of (g 0)
- $+(1, 2) = 3$
- call-by-need
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Need

- In the absence of side-effects, call-by-name and call-by-need always yield the same answer
- In the presence of side-effects, it is easy to distinguish them
- ```
  let g = let count = 0
          in proc (d)
           begin
                 set count = incr(count);
                 count
          end
  in (proc (x) +(x, x) (g 0) )
  ```
- g returns the number of times it is called
- Thunk for (g 0) is passed as the argument to the function in the body of the let
- call-by-name
- the first reference to x: sets count to 1 & returns 1 as the value of (g 0)
- the second reference to x: sets count to 2 & returns 2 as the value of (g 0)
- $+(1, 2) = 3$
- call-by-need
- the first reference to x forces: sets count to 1, returns 1 as the value of (g 0), and stores 1 as the value of (g 0)
  ```

Parameter Passing Variations

Lazy Evaluation: Call by Need

- In the absence of side-effects, call-by-name and call-by-need always yield the same answer
- In the presence of side-effects, it is easy to distinguish them
-
```
let g = let count = 0
        in proc (d)
         begin
               set count = incr(count);
               count
        end
in (proc (x) +(x, x) (g 0) )
```
- g returns the number of times it is called
- Thunk for (g 0) is passed as the argument to the function in the body of the let
- call-by-name
- the first reference to x: sets count to 1 & returns 1 as the value of (g 0)
- the second reference to x: sets count to 2 & returns 2 as the value of (g 0)
- $+(1, 2) = 3$
- call-by-need
- the first reference to x forces: sets count to 1, returns 1 as the value of (g 0), and stores 1 as the value of (g 0)
- second reference to x: returns the stored 1

Parameter Passing Variations

Lazy Evaluation: Call by Need

- In the absence of side-effects, call-by-name and call-by-need always yield the same answer
- In the presence of side-effects, it is easy to distinguish them
- ```
  let g = let count = 0
          in proc (d)
           begin
                 set count = incr(count);
                 count
          end
  in (proc (x) +(x, x) (g 0) )
  ```
- g returns the number of times it is called
- Thunk for (g 0) is passed as the argument to the function in the body of the let
- call-by-name
- the first reference to x: sets count to 1 & returns 1 as the value of (g 0)
- the second reference to x: sets count to 2 & returns 2 as the value of (g 0)
- $+(1, 2) = 3$
- call-by-need
- the first reference to x forces: sets count to 1, returns 1 as the value of (g 0), and stores 1 as the value of (g 0)
- second reference to x: returns the stored 1
- $+(1, 1) = 2$
  ```

Parameter Passing Variations

- Lazy evaluation: in the absence of side-effects allows for a simple way to reason about programs

Parameter Passing Variations

- Lazy evaluation: in the absence of side-effects allows for a simple way to reason about programs
- The effect of a procedure call is modeled by:
  - $\rightarrow$ Replacing the call with the body of the procedure
  - $\rightarrow$ Every reference to a parameter in the body is replaced by the corresponding operand
  - $\rightarrow$ This evaluation strategy is the basis of the lambda calculus and is known as β-reduction

Parameter Passing Variations

- Lazy evaluation: in the absence of side-effects allows for a simple way to reason about programs
- The effect of a procedure call is modeled by:
  - $\rightarrow$ Replacing the call with the body of the procedure
  - $\rightarrow$ Every reference to a parameter in the body is replaced by the corresponding operand
  - $\rightarrow$ This evaluation strategy is the basis of the lambda calculus and is known as β-reduction
- β-reduction: λ(x.e)x0 $\rightarrow$ e{x0/x}

```
λ(x.+(x, *(2, x)) -(5, -10)
→ +(-(5, -10) *(2, -(5, -10)))
→ +(15, *(2, -(5, -10)))
→ +(15, *(2, 15))
→ +(15, 30)
→ 45
```

Parameter Passing Variations

- All the freezing and thawing can lead to considerable overhead
- Reducing the number of thunks created is important for efficiency

Parameter Passing Variations

- All the freezing and thawing can lead to considerable overhead
- Reducing the number of thunks created is important for efficiency
- Difficult to determine the order of evaluation which is essential for programs with side-effects

Parameter Passing Variations

- All the freezing and thawing can lead to considerable overhead
- Reducing the number of thunks created is important for efficiency
- Difficult to determine the order of evaluation which is essential for programs with side-effects
- You do not have to think operationally: you can reason equationally about your programs.-–S. Doaitse Swierstra
- I prefer call by value to call by name because it is more predictable.-–Mitchell Wand

Parameter Passing Variations

- All the freezing and thawing can lead to considerable overhead
- Reducing the number of thunks created is important for efficiency
- Difficult to determine the order of evaluation which is essential for programs with side-effects
- You do not have to think operationally: you can reason equationally about your programs.-–S. Doaitse Swierstra
- I prefer call by value to call by name because it is more predictable.-–Mitchell Wand
- Popular with pure functional languages (i.e. with no side-effects) and rarely found elsewhere
- Haskell and Clean
- C# (deferred execution)

Parameter Passing Variations

- HOMEWORK: 4.31, 4.32, 4.39, 4.40, 4.42