#### End-of-semester evaluations

1. Please fill out the EOS evaluation.

1

1

# Lecture 23 Mutable Pairs – MP

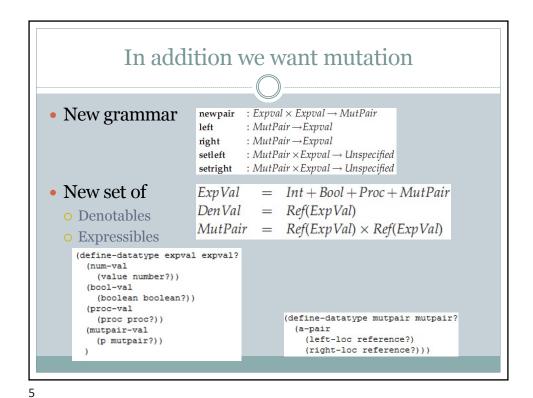
T. METIN SEZGIN

### Nugget

Now that we have a memory structure, we can add more sophisticated structures to our language

3

# Adding lists/pairs to the language



New scheme functions for pair management

```
right : MutPair → ExpVal
make-pair : ExpVal × ExpVal → MutPair
                                        (define right
(define make-pair
                                          (lambda (p)
  (lambda (vall val2)
                                            (cases mutpair p
                                              (a-pair (left-loc right-loc)
      (newref vall)
      (newref val2))))
                                                (deref right-loc)))))
                                       setleft : MutPair × ExpVal → Unspecified
left : MutPair → ExpVal
                                        (define setleft
(define left
  (lambda (p)
                                          (lambda (p val)
                                            (cases mutpair p
    (cases mutpair p
                                              (a-pair (left-loc right-loc)
      (a-pair (left-loc right-loc)
                                                (setref! left-loc val)))))
         (deref left-loc)))))
                                        setright : MutPair × ExpVal → Unspecified
                                        (define setright
                                          (lambda (p val)
                                            (cases mutpair p
(a-pair (left-loc right-loc)
                                                (setref! right-loc val)))))
```

#### The Interpreter (setleft-exp (expl exp2) (let ((vall (value-of expl env)) (val2 (value-of exp2 env))) (mutpair-val (make-pair vall val2)))) (let ((p (expval->mutpair vall))) (left-exp (expl) (setleft p val2) (let ((val1 (value-of exp1 env))) (let ((pl (expval->mutpair vall))) (num-val 82))))) (left pl)))) (setright-exp (expl exp2) (let ((vall (value-of expl env)) (right-exp (expl) (val2 (value-of exp2 env))) (let ((vall (value-of expl env))) (let ((pl (expval->mutpair vall))) (right pl)))) (let ((p (expval->mutpair vall))) (begin (setright p val2) (num-val 83)))))

7

#### Nugget

We can get creative and devise a more efficient implementation

#### A different representation for mutable pairs

 Note something about the addresses of the two values

9

#### A different representation for mutable pairs

### Learning outcomes of this lecture

- A student attending this lecture should be able to:
  - 1. Understand how pairs can be implemented, and do so
  - 2. Explain why the second implementation is more efficient
  - 3. Implement more sophisticated data structures (e.g., stack, arrays).

11

# Lecture 24 Parameter Passing

T. METIN SEZGIN

# Learning outcomes of this lecture

- A student attending this lecture should be able to:
  - 1. Understand that there are variations to parameter passing
  - 2. Understand CBV/CBR and how they work
  - 3. Understand the uses of CBR
  - 4. Trace and CBV/CBR evaluation using the env & store
  - 5. Implement CBR/CBR

13

#### Nugget

There are flavors to parameter passing.

What is the value of the following expression?

• What happens during evaluation?

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

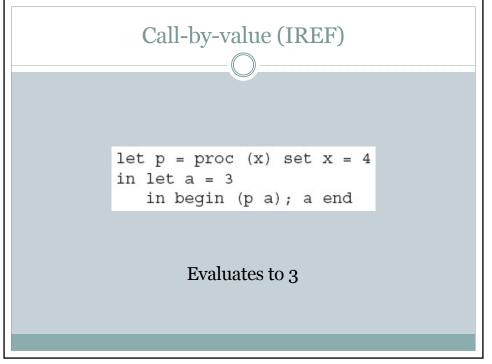
15

#### **Parameter Passing Variations**

- Natural (PROC)
- Call-by-value
- Call-by-reference
- Call-by-name (lazy evaluation)
- Call-by-need (lazy evaluation)

# PROC let p = proc (x) set x = 4 in let a = 3 in begin (p a); a end Evaluates to 3

17



# IREF -- Call-by-reference

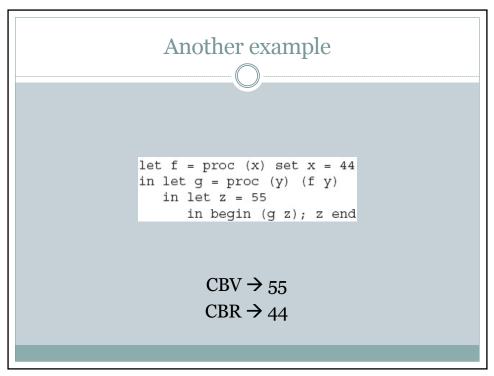
let p = proc (x) set x = 4
in let a = 3
 in begin (p a); a end

Evaluates to 4

19

# Nugget

In Call by Value, a copy of the argument is passed



```
let f = proc(x) set x = 44
in let g = proc (y) (f y)
in let z = 55
                                                                                        Evaluation trace
in begin
        (g z);
     end
  > (run "
let f = proc (x) set x = 44
in let g = proc (y) (f y)
in let z = 55
                                                                                                                                        entering let g
newref: allocating location 4
entering body of let g with env =
((g 4) (f 3) (i 0) (v 1) (x 2))
store =
  in begin (g z);
                                                                                                                                         ((0 #(struct:num-val 1))
                                                                                                                                           ((0 #(struct:num-val 1))
(1 #(struct:num-val 5))
(2 #(struct:num-val 10))
(3 (procedure x ... ((i 0) (v 1) (x 2))))
(4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
       end")
  newref: allocating location 0
newref: allocating location 1
newref: allocating location 2
 newref: allocating location 2
entering let f
newref: allocating location 3
entering body of let f with env =
((f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
(1 #(struct:num-val 5))
(2 #(struct:num-val 10))
(3 (procedure x ... ((i 0) (v 1) (x 2)))))
                                                                                                                                        entering let z
newref: allocating location 5
entering body of let z with env =
((z 5) (g 4) (f 3) (i 0) (v 1) (x 2))
store =
                                                                                                                                         ((0 #(struct:num-val 1))
(1 #(struct:num-val 5))
                                                                                                                                           (1 #(struct:num-val 5))
(2 #(struct:num-val 10))
(3 (procedure x ... ((i 0) (v 1) (x 2))))
(4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
(5 #(struct:num-val 55)))
```

#### 

# Learning outcomes of this lecture

- A student attending this lecture should be able to:
  - 1. Understand that there are variations to parameter passing
  - 2. Understand CBV/CBR and how they work
  - 3. Understand the uses of CBR
  - 4. Trace and CBV/CBR evaluation using the env & store
  - 5. Implement CBR

25

Questions?



- Expressed and denoted values remain the same ExpVal = Int + Bool + ProcDenVal = Ref(ExpVal)
- Location allocation policy changes
  - If the formal parameter is a variable, pass on the reference
  - Otherwise, put the value of the formal parameter into the memory, pass a reference to it

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

#### Another example

- Here there is variable aliasing
- This evaluates to 4

### Lazy evaluation

- Call-by-name
- Call-by-need

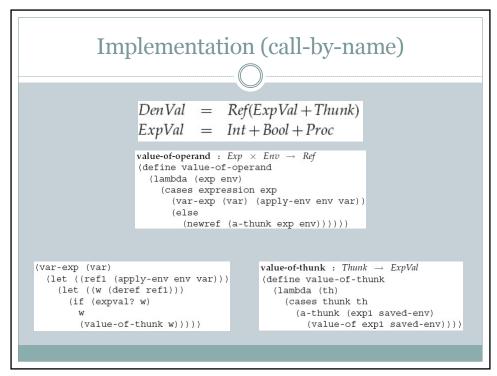
```
letrec infinite-loop (x) = infinite-loop(-(x,-1))
in let f = proc (z) 11
  in (f (infinite-loop 0))
```

29

#### Thunks

• Save any future work for the future

```
(define-datatype thunk thunk?
  (a-thunk
    (exp1 expression?)
    (env environment?)))
```



# Quiz #8

• In the previous lecture we introduced two implementations for mutable pairs – the second one being a more space-efficient version. The implementation for setleft is given below. Write down the implementation for setright.