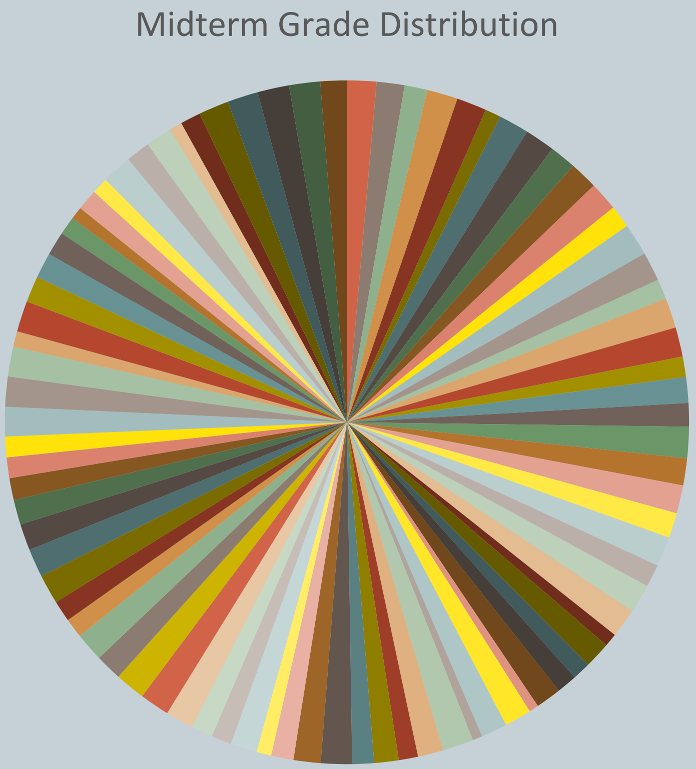
IREF - Review

T. METIN SEZGIN

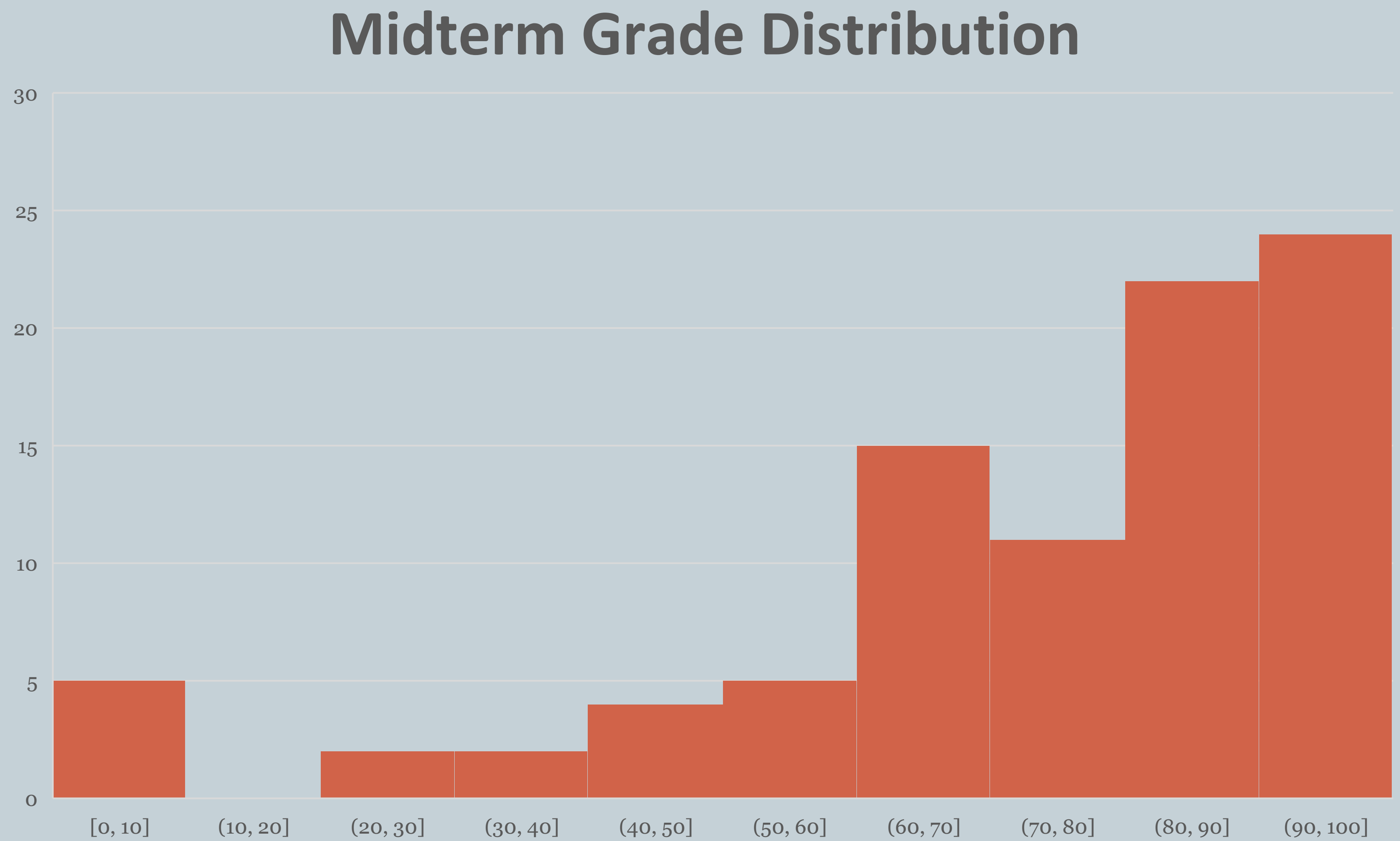
Announcements

- Project 3 out
- Midterm grade statistics today

Midterm Statistics



Midterm Statistics



Midterm Statistics

	Average Grades
Group A	69.1
Group B	83.5

Midterm Statistics

	Average Grades
Group A	41.2
Group B	56.4
Group C	71.6
Group D	81.7
Group E	88.8

Midterm Statistics

	Average Grades
Group A	41.2
Group B	53.6
Group C	70.2
Group D	80.2
Group E	87.0
Group F	74
Group G	75.4
Group H	86.5
Group I	90.8

Midterm Statistics

	Average Grades
Group A	69.1
Group B	83.5

Midterm Statistics

Took notes?	Average Grades
No	69.1
Yes	83.5

Midterm Statistics

	Average Grades
Group A	41.2
Group B	56.4
Group C	71.6
Group D	81.7
Group E	88.8

Midterm Statistics

Attendance	Average Grades
%0-%20	41.2
%20-%40	56.4
%40-%60	71.6
%60-%80	81.7
%80-%20	88.8

Midterm Statistics

	Average Grades
Group A	41.2
Group B	53.6
Group C	70.2
Group D	80.2
Group E	87.0
Group F	74
Group G	75.4
Group H	86.5
Group I	90.8

Midterm Statistics

PL-score	Average Grades
0	41.2
1	53.6
2	70.2
3	80.2
4	87.0
5	74
6	75.4
7	86.5
8	90.8

PL-score: 4xL + P

Where **L** = 1 if took lecture notes, 0 otherwise

P = Presence quartile (0 if 0%-20%,

1 if 20%-40%, …)

Questions?

Review

ExpVal = Int + Bool + Proc

DenVal = Ref (ExpVal)

set a = 9

↳ reference to a memory location

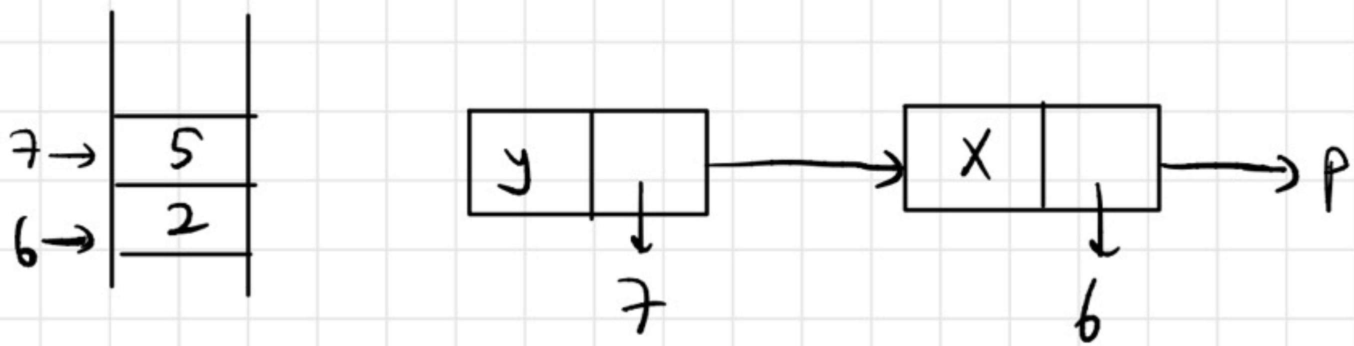
Grammar

Expression ::= set Identifier = Expression

$\boxed{\text{assign-exp (var exp1)}}$

Bartu Uzun

Ex: let x=2
in let y=5
in $\boxed{-(x,y)}$ → What does the env and memory look like when I evaluate this?



↳ There will be similar questions in quizzes and exam

Eren Ceylan

Must note any potential change to the memory.

$$\frac{\begin{array}{c}(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (val_1, \sigma_1)\\ (\texttt{value-of}\ exp_2\ \rho\ \sigma_1) = (val_2, \sigma_2)\end{array}}{(\texttt{value-of}\ (\texttt{diff-exp}\ exp_1\ exp_2)\ \rho\ \sigma_0) = (\lceil\lfloor val_1\rfloor - \lfloor val_2\rfloor\rceil, \sigma_2)}$$

In case exp1 changes the memory, must use sigma 1, not 0

$$\frac{(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (val_1, \sigma_1)}{\begin{array}{l}(\texttt{value-of}\ (\texttt{if-exp}\ exp_1\ exp_2\ exp_3)\ \rho\ \sigma_0)\\ \quad = \begin{cases}(\texttt{value-of}\ exp_2\ \rho\ \sigma_1) & \text{if}\ (\texttt{expval->bool}\ val_1) = \#\text{t}\\ (\texttt{value-of}\ exp_3\ \rho\ \sigma_1) & \text{if}\ (\texttt{expval->bool}\ val_1) = \#\text{f}\end{cases}\end{array}}$$

Again, must use new sigma just in case the memory is changed

Unal Cama

IREF & Mutable Pairs

T. METIN SEZGIN

Implicit references

- IREF
  - References are instantiated by the interpreter
  - All denoted values are references to expressed values
  - Each binding operation introduces a location
    - Let
    - letrec
    - proc
  - Pointers to stores are saved in the environment

$$\begin{aligned} ExpVal &= Int + Bool + Proc \\ DenVal &= Ref(ExpVal) \end{aligned}$$

New grammar

- A set operation for assignment

$$\mathit{Expression} ::= \texttt{set}\ \mathit{Identifier}\ \texttt{=}\ \mathit{Expression}$$
$$\boxed{\texttt{assign-exp (var exp1)}}$$

Examples

```
let x = 0
in letrec even(dummy)
           = if zero?(x)
             then 1
             else begin
                   set x = -(x,1);
                   (odd 888)
                  end
          odd(dummy)
           = if zero?(x)
             then 0
             else begin
                   set x = -(x,1);
                   (even 888)
                  end
   in begin set x = 13; (odd -888) end
```

```
let g = let count = 0
        in proc (dummy)
            begin
             set count = -(count,-1);
             count
            end
in let a = (g 11)
   in let b = (g 11)
      in -(a,b)
```

Behavior specification

- **`var-exp`**

$$(\texttt{value-of}\ (\texttt{var-exp}\ var)\ \rho\ \sigma) = (\sigma(\rho(var)), \sigma)$$

- **`assign-exp`**

$$\frac{(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (val_1, \sigma_1)}{(\texttt{value-of}\ (\texttt{assign-exp}\ var\ exp_1)\ \rho\ \sigma_0) = (\lceil 27 \rceil, [\rho(var) = val_1]\sigma_1)}$$

- **`apply-procedure`**

$$\begin{array}{l}(\texttt{apply-procedure}\ (\texttt{procedure}\ var\ body\ \rho)\ val\ \sigma) \\ = (\texttt{value-of}\ body\ [var = l]\rho\ [l = val]\sigma)\end{array}$$

Implementation

- **var-exp**

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

- **assign-exp**

```
(assign-exp (var exp1)
  (begin
    (setref!
      (apply-env env var)
      (value-of exp1 env))
    (num-val 27)))
```

- **apply-procedure**

apply-procedure : $\mathit{Proc} \times \mathit{ExpVal} \to \mathit{ExpVal}$

```
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body
          (extend-env var (newref val) saved-env))))))
```

Implementation

Reference instantiations

- **apply-procedure**

apply-procedure : *Proc* × *ExpVal* → *ExpVal*

```
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body
          (extend-env var (newref val) saved-env))))))
```

- **let**

```
(let-exp (var exp1 body)
  (let ((val1 (value-of exp1 env)))
    (value-of body
      (extend-env var (newref val1) env))))
```

- **letrec**

```
(extend-env-rec (p-names b-vars p-bodies saved-env)
  (let ((n (location search-var p-names)))
    (if n
      (newref
        (proc-val
          (procedure
            (list-ref b-vars n)
            (list-ref p-bodies n)
            env)))
      (apply-env saved-env search-var))))
```

Mutable Pairs

T. METIN SEZGIN

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 alternative implementations of pairs
  3. Implement more sophisticated data structures (e.g., stack, arrays).

Nugget

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

Adding lists/pairs to the language

```
let x = 4
in cons(x,
        cons(cons(-(x,1),
                  emptylist),
             emptylist))
```

Nugget

Having a memory feature allows us to have

mutable pairs

In addition we want mutation

- New grammar

newpair	: $Expval \times Expval \to MutPair$
left	: $MutPair \to Expval$
right	: $MutPair \to Expval$
setleft	: $MutPair \times Expval \to Unspecified$
setright	: $MutPair \times Expval \to Unspecified$

- New set of
  - Denotables
  - Expressibles

$$
\begin{aligned}
ExpVal &= Int + Bool + Proc + MutPair \\
DenVal &= Ref(ExpVal) \\
MutPair &= Ref(ExpVal) \times Ref(ExpVal)
\end{aligned}
$$

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

```
(define-datatype mutpair mutpair?
  (a-pair
    (left-loc reference?)
    (right-loc reference?)))
```

New scheme functions for pair management

make-pair : *ExpVal* × *ExpVal* → *MutPair*

```
(define make-pair
  (lambda (val1 val2)
    (a-pair
      (newref val1)
      (newref val2))))
```

left : *MutPair* → *ExpVal*

```
(define left
  (lambda (p)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (deref left-loc)))))
```

right : *MutPair* → *ExpVal*

```
(define right
  (lambda (p)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (deref right-loc)))))
```

setleft : *MutPair* × *ExpVal* → *Unspecified*

```
(define setleft
  (lambda (p val)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (setref! left-loc val)))))
```

setright : *MutPair* × *ExpVal* → *Unspecified*

```
(define setright
  (lambda (p val)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (setref! right-loc val)))))
```

The Interpreter

```
(newpair-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (mutpair-val (make-pair val1 val2))))

(left-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((p1 (expval->mutpair val1)))
      (left p1))))

(right-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((p1 (expval->mutpair val1)))
      (right p1))))
```

```
(setleft-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (let ((p (expval->mutpair val1)))
      (begin
        (setleft p val2)
        (num-val 82)))))

(setright-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (let ((p (expval->mutpair val1)))
      (begin
        (setright p val2)
        (num-val 83)))))
```

Nugget

We can get creative and devise a more efficient implementation

A different representation for mutable pairs

make-pair : *ExpVal* × *ExpVal* → *MutPair*
```
(define make-pair
  (lambda (val1 val2)
    (a-pair
      (newref val1)
      (newref val2))))
```

left : *MutPair* → *ExpVal*
```
(define left
  (lambda (p)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (deref left-loc)))))
```

- Note something about the addresses of the two values

A different representation for mutable pairs

mutpair? : *SchemeVal* → *Bool*

```
(define mutpair?
  (lambda (v)
    (reference? v)))
```

make-pair : *ExpVal* × *ExpVal* → *MutPair*

```
(define make-pair
  (lambda (val1 val2)
    (let ((ref1 (newref val1)))
      (let ((ref2 (newref val2)))
        ref1))))
```

left : *MutPair* → *ExpVal*

```
(define left
  (lambda (p)
    (deref p)))
```

right : *MutPair* → *ExpVal*

```
(define right
  (lambda (p)
    (deref (+ 1 p))))
```

setleft : *MutPair* × *ExpVal* → *Unspecified*

```
(define setleft
  (lambda (p val)
    (setref! p val)))
```

setright : *MutPair* × *ExpVal* → *Unspecified*

```
(define setright
  (lambda (p val)
    (setref! (+ 1 p) val)))
```

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 alternative implementations of pairs
  3. Implement more sophisticated data structures (e.g., stack, arrays).