Announcements

1. No physical lectures next week – online only
2. Mid-semester evaluations, suggestions:
    1. Quizzes made longer
    2. Quizzes made more basic
    3. Additional quiz for people w/excuse
    4. More points to labs

Announcements

1. No physical lectures next week – online only
2. Mid-semester evaluations, suggestions:
    1. ✓ Quizzes made longer
    2. ✓ Quizzes made more basic
    3. ✓ Additional quiz for people w/excuse
    4. ✓ More points to labs

Announcements

1. No physical lectures next week – online only
2. Mid-semester evaluations, suggestions:
    1. ✓ Quizzes made longer
    2. ✓ Quizzes made more basic
    3. ✓ Additional quiz for people w/excuse
    4. ✓ More points to labs
3. Unjustified criticism

Lecture 13 – Review
Let – Implementation

T. METIN SEZGIN

Behavior implementation

what we envision

Let ρ = [x=⌈33⌉,y=⌈22⌉].

```
(value-of
  <<if zero?(-(x,11)) then -(y,2) else -(y,4)>>
  ρ)

= (if (expval->bool (value-of <<zero?(-(x,11))>> ρ))
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (if (expval->bool (bool-val #f))
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (if #f
    (value-of <<-(y,2)>> ρ)
    (value-of <<-(y,4)>> ρ))

= (value-of <<-(y,4)>> ρ)

= ⌈18⌉
```

Nugget

Intro to implementation

It all revolves around **`value-of`**

The Interpreter

```
run : String → ExpVal
(define run
  (lambda (string)
    (value-of-program (scan&parse string))))

value-of-program : Program → ExpVal
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (value-of exp1 (init-env))))))
```

The Interpreter

```
value-of : Exp × Env → ExpVal
(define value-of
  (lambda (exp env)
    (cases expression exp
```

$$\frac{}{\texttt{(value-of (const-exp } n\texttt{) } \rho\texttt{)} = n}$$

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

$$\frac{}{\texttt{(value-of (var-exp } var\texttt{) } \rho\texttt{)} = \texttt{(apply-env } \rho\ var\texttt{)}}$$

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

$$\frac{}{\texttt{(value-of (diff-exp } exp_1\ exp_2\texttt{) } \rho\texttt{)} = \lceil\texttt{(-} \lfloor\texttt{(value-of } exp_1\ \rho\texttt{)}\rfloor\ \lfloor\texttt{(value-of } exp_2\ \rho\texttt{)}\rfloor\texttt{)}\rceil}$$

```
(diff-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (let ((num1 (expval->num val1))
          (num2 (expval->num val2)))
      (num-val
        (- num1 num2)))))
```

$$\frac{\texttt{(value-of } exp_1\ \rho\texttt{)} = val_1}{\texttt{(value-of (zero?-exp } exp_1\texttt{) } \rho\texttt{)} = \begin{cases}\texttt{(bool-val \#t)} & \text{if } \texttt{(expval->num } val_1\texttt{)} = 0\\ \texttt{(bool-val \#f)} & \text{if } \texttt{(expval->num } val_1\texttt{)} \neq 0\end{cases}}$$

```
(zero?-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((num1 (expval->num val1)))
      (if (zero? num1)
        (bool-val #t)
        (bool-val #f)))))
```

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

```
(if-exp (exp1 exp2 exp3)
  (let ((val1 (value-of exp1 env)))
    (if (expval->bool val1)
      (value-of exp2 env)
      (value-of exp3 env))))
```

$$\frac{\texttt{(value-of } exp_1\ \rho\texttt{)} = val_1}{\texttt{(value-of (let-exp } var\ exp_1\ body\texttt{) } \rho\texttt{)} = \texttt{(value-of } body\ [var = val_1]\rho\texttt{)}}$$

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

Lecture 14
PROC

T. METIN SEZGIN

LET is ex; long live PROC

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

Expressed and Denoted values

- Before

$$ExpVal = Int + Bool$$
$$DenVal = Int + Bool$$

- After

$$ExpVal = Int + Bool + Proc$$
$$DenVal = Int + Bool + Proc$$

Examples

Expression ::= `proc` (*Identifier*) *Expression*

```
proc-exp (var body)
```

Expression ::= (*Expression* *Expression*)

```
call-exp (rator rand)
```

- Concepts
  - In definition
    - var
      - Bound variable (a.k.a. formal parameter)
  - In procedure call
    - Rand
      - Actual parameter (the value → argument)
    - Rator
      - Operator

5+3×2

Syntax for constructing and calling procedures

Expression ::= `proc (`*Identifier*`)` *Expression*
`proc-exp (var body)`

Expression ::= `(`*Expression* *Expression*`)`
`call-exp (rator rand)`

```
let f = proc (x) -(x,11)
in (f (f 77))

(proc (f) (f (f 77))
 proc (x) -(x,11))
```

Syntax for constructing and calling procedures

Expression ::= `proc (`*Identifier*`)` *Expression*
`proc-exp (var body)`

Expression ::= `(`*Expression* *Expression*`)`
`call-exp (rator rand)`

```
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))
```

The interface for PROC

- Procedures have
  - Constructor → **procedure**

```
(value-of (proc-exp var body) ρ)
= (proc-val (procedure var body ρ))
```

  - Observer → **apply-procedure**

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

The intuition behind application

- Extend the environment
- Evaluate the body

```
(apply-procedure (procedure var body ρ) val)
= (value-of body [var=val]ρ)
```

```
(value-of
  <<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))>>
  ρ)

= (value-of
    <<let f = proc (z) -(z,x)
      in let x = 100
         in let g = proc (z) -(z,x)
            in -((f 1), (g 1))>>
    [x=⌈200⌉]ρ)


= (value-of
    <<let x = 100
      in let g = proc (z) -(z,x)
         in -((f 1), (g 1))>>
    [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
     [x=⌈200⌉]ρ)

= (value-of
    <<let g = proc (z) -(z,x)
      in -((f 1), (g 1))>>
    [x=⌈100⌉]
     [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
      [x=⌈200⌉]ρ)
```

```
= (value-of
    <<-((f 1), (g 1))>>
    [g=(proc-val (procedure z <<-(z,x)>>
                   [x=⌈100⌉][f=...][x=⌈200⌉]ρ))]
     [x=⌈100⌉]
      [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
       [x=⌈200⌉]ρ)

= ⌈(-
    (value-of <<(f 1)>>
      [g=(proc-val (procedure z <<-(z,x)>>
                     [x=⌈100⌉][f=...][x=⌈200⌉]ρ))]
       [x=⌈100⌉]
        [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
         [x=⌈200⌉]ρ)
    (value-of <<(g 1)>>
      [g=(proc-val (procedure z <<-(z,x)>>
                     [x=⌈100⌉][f=...][x=⌈200⌉]ρ))]
       [x=⌈100⌉]
        [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉]ρ))]
         [x=⌈200⌉]ρ))⌉

= ⌈(-
    (apply-procedure
      (procedure z <<-(z,x)>> [x=⌈200⌉]ρ)
      ⌈1⌉)
    (apply-procedure
      (procedure z <<-(z,x)>> [x=⌈100⌉][f=...][x=⌈200⌉]ρ)
      ⌈1⌉))⌉
```

An example

```
= ⌈(-
    (value-of <<(f 1)>>
      [g=(proc-val (procedure z <<-(z,x)>>
                     [x=⌈100⌉][f=...][x=⌈200⌉ρ))]
       [x=⌈100⌉]
        [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉ρ))]
         [x=⌈200⌉ρ)
    (value-of <<(g 1)>>
      [g=(proc-val (procedure z <<-(z,x)>>
                     [x=⌈100⌉][f=...][x=⌈200⌉ρ))]
       [x=⌈100⌉]
        [f=(proc-val (procedure z <<-(z,x)>> [x=⌈200⌉ρ))]
         [x=⌈200⌉ρ))⌉

= ⌈(-
    (apply-procedure
      (procedure z <<-(z,x)>> [x=⌈200⌉ρ)
      ⌈1⌉)
    (apply-procedure
      (procedure z <<-(z,x)>> [x=⌈100⌉][f=...][x=⌈200⌉ρ)
      ⌈1⌉))⌉

= ⌈(-
    (value-of <<-(z,x)>> [z=⌈1⌉][x=⌈200⌉ρ)
    (value-of <<-(z,x)>> [z=⌈1⌉][x=⌈100⌉][f=...][x=⌈200⌉ρ))⌉

= ⌈(- -199 -99)⌉

= ⌈-100⌉
```

Implementation

proc? : *SchemeVal* → *Bool*

```
(define proc?
  (lambda (val)
    (procedure? val)))
```

procedure : *Var* × *Exp* × *Env* → *Proc*

```
(define procedure
  (lambda (var body env)
    (lambda (val)
      (value-of body (extend-env var val env)))))
```

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

```
(define apply-procedure
  (lambda (proc1 val)
    (proc1 val)))
```

Alternative implementation

proc? : *SchemeVal* → *Bool*
procedure : *Var* × *Exp* × *Env* → *Proc*

```
(define-datatype proc proc?
  (procedure
    (var identifier?)
    (body expression?)
    (saved-env environment?)))

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

Other changes to the interpreter

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


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

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