Lecture 12 – Review
Let – Implementation
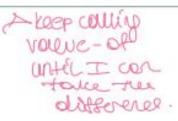
T. METIN SEZGIN

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))))))
```

Behavior implementation

what we envision

Let $\rho = [i=1, v=5, x=10]$.

```
(value-of
  <<- (- (x,3), - (v,i) ) >>
  ρ)
```

keep calling value of subparts.

$$= \lceil (- $$
```
  ⌊(value-of <<- (x,3) >> ρ)⌋
  ⌊(value-of <<- (v,i) >> ρ)⌋) ⌉
```

$$= \lceil (- $$
```
  (-
    ⌊(value-of <<x>> ρ)⌋
    ⌊(value-of <<3>> ρ)⌋)
  ⌊(value-of <<- (v,i) >> ρ)⌋) ⌉
```

$$= \lceil (- $$
```
  (-
    10   value of x
    ⌊(value-of <<3>> ρ)⌋)
  (value-of <<- (v,i) >> ρ)) ⌉
```

$$= \lceil (- $$
```
    (-
      10
      3)
    ⌊(value-of <<- (v,i) >> ρ)⌋) ⌉
```

$$= \lceil (- $$
```
    7
    ⌊(value-of <<- (v,i) >> ρ)⌋) ⌉
```

evaluating $- (x,3)$

$$= \lceil (- $$
```
    7
    (-
      ⌊(value-of <<v>> ρ)⌋
      ⌊(value-of <<i>> ρ)⌋)) ⌉
```

$$= \lceil (- $$
```
    7
    (-      value of v
      5
      ⌊(value-of <<i>> ρ)⌋)) ⌉
```

$$= \lceil (- $$
```
    7
    (-
      value 5
      of ( ↘1))⌉
```

$$= \lceil (- $$
```
    7
    -(v,i) → 4)⌉
```

$$= \lceil 3 \rceil$$

interpreter's representation

Duru Tandogan

The Interpreter

value-of : $Exp \times Env \rightarrow ExpVal$
```
(define value-of
  (lambda (exp env)
    (cases expression exp
```

$$(\text{value-of} \ (\text{const-exp} \ n) \ \rho) = n$$
```
(const-exp (num) (num-val num))
```

$$(\text{value-of} \ (\text{var-exp} \ var) \ \rho) = (\text{apply-env} \ \rho \ var)$$
```
(var-exp (var) (apply-env env var))
```

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

$$\frac{(\text{value-of} \ exp_1 \ \rho) = val_1}{(\text{value-of} \ (\text{zero?-exp} \ exp_1) \ \rho) = \begin{cases} (\text{bool-val} \ \#t) & \text{if} \ (\text{expval->num} \ val_1) = 0 \\ (\text{bool-val} \ \#f) & \text{if} \ (\text{expval->num} \ val_1) \neq 0 \end{cases}}$$
```
(zero?-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((num1 (expval->num val1)))
      (if (zero? num1)
        (bool-val #t)
        (bool-val #f)))))
```

$$\frac{(\text{value-of} \ exp_1 \ \rho) = val_1}{(\text{value-of} \ (\text{if-exp} \ exp_1 \ exp_2 \ exp_3) \ \rho) = \begin{cases} (\text{value-of} \ exp_2 \ \rho) & \text{if} \ (\text{expval->bool} \ val_1) = \#t \\ (\text{value-of} \ exp_3 \ \rho) & \text{if} \ (\text{expval->bool} \ val_1) = \#f \end{cases}}$$
```
(if-exp (exp1 exp2 exp3)
  (let ((val1 (value-of exp1 env)))
    (if (expval->bool val1)
      (value-of exp2 env)
      (value-of exp3 env))))
```

$$\frac{(\text{value-of} \ exp_1 \ \rho) = val_1}{(\text{value-of} \ (\text{let-exp} \ var \ exp_1 \ body) \ \rho) = (\text{value-of} \ body \ [var = val_1]\rho)}$$
```
(let-exp (var exp1 body)
  (let ((val1 (value-of exp1 env)))
    (value-of body
      (extend-env var val1 env))))))))
```

Lecture 13
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

Syntax for constructing and calling procedures

$$Expression ::= \text{proc } (Identifier) \; Expression$$

```
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)))
```