Lecture 12 – Review
Let – Implementation

T. METIN SEZGIN

Lecture Notes

Evaluating a value always creates an evaluated value

$$\frac{(\texttt{value-of}\ \mathit{exp}_1\ \ \rho) = \mathit{val}_1}{\begin{array}{l}(\texttt{value-of}\ (\texttt{zero?-exp}\ \mathit{exp}_1)\ \ \rho) \\ \quad = \begin{cases}(\texttt{bool-val}\ \texttt{\#t}) & \text{if } (\texttt{expval->num}\ \mathit{val}_1) = 0 \\ (\texttt{bool-val}\ \texttt{\#f}) & \text{if } (\texttt{expval->num}\ \mathit{val}_1) \neq 0\end{cases}\end{array}}$$

zero? - exp

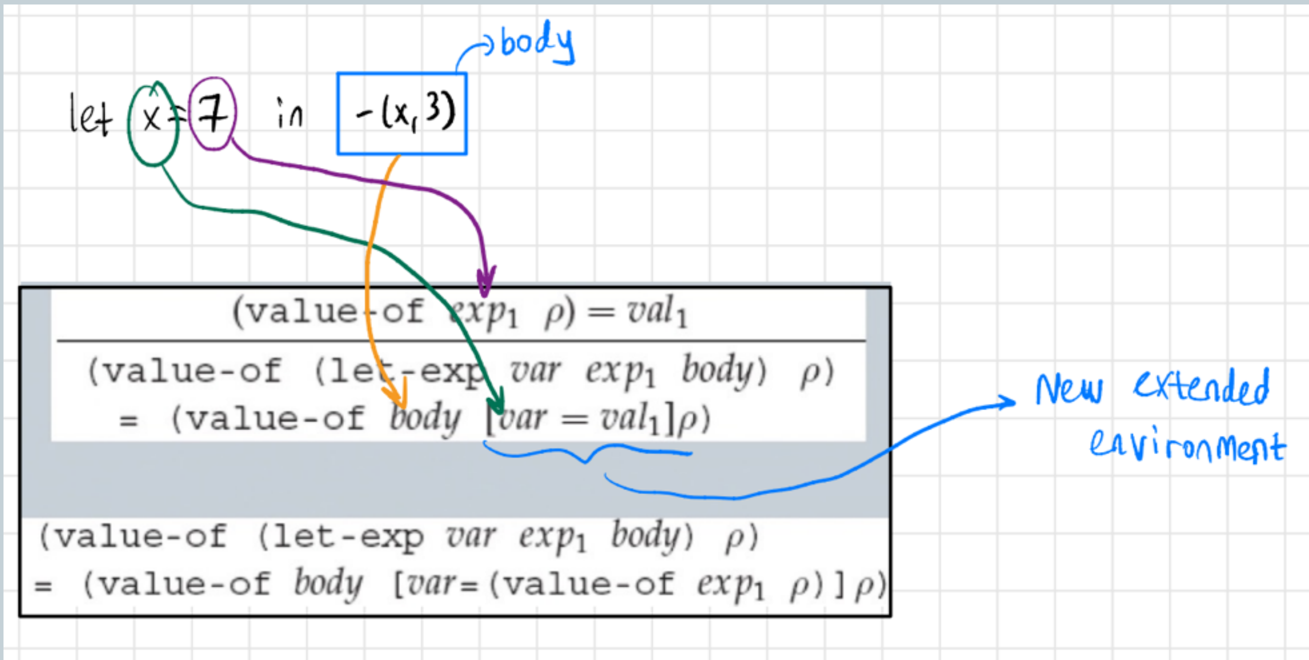
specifying the behavior

$$\frac{(\texttt{value-of}\ \mathit{exp}_1\ \ \rho) = \mathit{val}_1}{\begin{array}{l}(\texttt{value-of}\ (\texttt{if-exp}\ \mathit{exp}_1\ \mathit{exp}_2\ \mathit{exp}_3)\ \ \rho) \\ \quad = \begin{cases}(\texttt{value-of}\ \mathit{exp}_2\ \ \rho) & \text{if } (\texttt{expval->bool}\ \mathit{val}_1) = \texttt{\#t} \\ (\texttt{value-of}\ \mathit{exp}_3\ \ \rho) & \text{if } (\texttt{expval->bool}\ \mathit{val}_1) = \texttt{\#f}\end{cases}\end{array}}$$

if-exp

Eren Ceylan

Lecture Notes



Eren Ceylan

Lecture Notes

Behavior Implementation

Let $\rho =$ `[i=1,v=5,x=10].` → initial environment

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

= ⌈(-
    ⌊(value-of <<-(x,3)>> ρ)⌋
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

= ⌈(-
    (-
     ⌊(value-of <<x>> ρ)⌋
     ⌊(value-of <<3>> ρ)⌋)
    ⌊(value-of <<-(v,i)>> ρ)⌋)⌉

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

produces expval of 7

according to behavioral specification "x" will be searched in the env. (ρ)

↳ produces expVal

<< exp >>

↓

Abstract Syntax Tree

↳ Possible Quiz Question!

Lecture Notes

Remark 1:

Interpreters are architecture specific.
mapping an interpreter to another one is simple when we have the interfaces

[redacted]

value-of-program

doesn't always have to be empty.

$$(\text{value\_of} \quad \text{exp} \quad \underbrace{[\, i = [1],\ v = [5],\ x = [10] \,]}_{\text{environment}})$$

Farrin Sofian

Lecture Notes

How constructors are used:

```
(value-of (const-exp n) ρ) = (num-val n)
(value-of (var-exp var) ρ) = (apply-env ρ var)

(value-of (diff-exp exp1 exp2) ρ)
= (num-val → convert back to expval.
    (-
      (expval->num (value-of exp1 ρ))
      (expval->num (value-of exp2 ρ))))
```

▷ machine representation why needed? because Value-of always returns expval, hence we need to revert back to machine representation

body

Farrin Sofian

: Some operations need to be supported by the machine itself.
: value-of called recursively
: exp types.

The Interpreter

why cases? → we have diff exps,
Cases checks which exp we have got.

value-of : $Exp \times Env \to ExpVal$

```
(define value-of
  (lambda (exp env)   ↗ type expression.
    (cases expression exp
```

Behaviour specification

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

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

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

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

Name of the identifier

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

```
      (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)))))
```

Computer representation

let $\boxed{x}$ = $\boxed{-(10, 7)}$ (exp1)
var
in $-(x, 2)$
Body

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

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

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

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

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

```
      (let-exp (var exp1 body)   3
        (let ((val1 (value-of exp1 env)))
          (value-of body
            (extend-env var val1 env)))))))
```

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 : $Exp \times Env \rightarrow ExpVal$

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

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

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

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

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

$$\boxed{\begin{array}{l}\texttt{(value-of (diff-exp } exp_1\ exp_2\texttt{) } \rho\texttt{) =} \\ \quad \lceil\texttt{(- } \lfloor\texttt{(value-of } exp_1\ \rho\texttt{)}\rfloor\ \lfloor\texttt{(value-of } exp_2\ \rho\texttt{)}\rfloor\texttt{)}\rceil\end{array}}$$

```
      (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)))))
```

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

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

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

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

$$\boxed{\dfrac{\texttt{(value-of } exp_1\ \rho\texttt{)} = val_1}{\begin{array}{l}\texttt{(value-of (let-exp } var\ exp_1\ body\texttt{) } \rho\texttt{)} \\ \quad = \texttt{(value-of } body\ [var = val_1]\rho\texttt{)}\end{array}}}$$

```
(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

$$Expression ::= \texttt{proc}\ (Identifier)\ Expression$$
$$\boxed{\texttt{proc-exp (var body)}}$$

$$Expression ::= (Expression\ \ Expression)$$
$$\boxed{\texttt{call-exp (rator rand)}}$$

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

5+ 3 x 2

Syntax for constructing and calling procedures

$$Expression ::= \texttt{proc}\ \ (Identifier)\ \ Expression$$

$$\boxed{\texttt{proc-exp (var body)}}$$

$$Expression ::= (Expression\ \ Expression)$$

$$\boxed{\texttt{call-exp (rator rand)}}$$

```
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

$$Expression ::= \texttt{proc}\ \ (Identifier)\ \ Expression$$
$$\boxed{\texttt{proc-exp (var body)}}$$

$$Expression ::= (Expression\ \ Expression)$$
$$\boxed{\texttt{call-exp (rator rand)}}$$

```
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`**

$$\texttt{(value-of (proc-exp } \mathit{var}\ \mathit{body}\texttt{)}\ \rho\texttt{)}$$
$$\texttt{= (proc-val (procedure } \mathit{var}\ \mathit{body}\ \rho\texttt{))}$$

  - Observer → **`apply-procedure`**

$$\texttt{(value-of (call-exp } \mathit{rator}\ \mathit{rand}\texttt{)}\ \rho\texttt{)}$$
$$\texttt{= (let ((proc (expval->proc (value-of } \mathit{rator}\ \rho\texttt{)))}$$
$$\texttt{(arg (value-of } \mathit{rand}\ \rho\texttt{)))}$$
$$\texttt{(apply-procedure proc arg))}$$

The intuition behind application

- Extend the environment
- Evaluate the body

$$
\begin{aligned}
&\texttt{(apply-procedure (procedure } \mathit{var}\ \mathit{body}\ \rho\texttt{)}\ \mathit{val}\texttt{)}\\
&= \texttt{(value-of } \mathit{body}\ [\mathit{var}{=}\mathit{val}]\rho\texttt{)}
\end{aligned}
$$

```
(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? : $SchemeVal \rightarrow Bool$

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

procedure : $Var \times Exp \times Env \rightarrow Proc$

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

apply-procedure : $Proc \times ExpVal \rightarrow ExpVal$

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

Alternative implementation

proc? : $SchemeVal \rightarrow Bool$
procedure : $Var \times Exp \times Env \rightarrow Proc$

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

apply-procedure : $Proc \times ExpVal \rightarrow ExpVal$

```
(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)))
```