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))))))
```

The Interpreter

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

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

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

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

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

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

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

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

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

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

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

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

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

```
      (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

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

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

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

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

```
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

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

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

$$\mathit{Expression} ::= (\mathit{Expression}\ \ \mathit{Expression})$$

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

```
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

(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? : $SchemeVal \rightarrow Bool$

```
(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)))
```