

Lecture 13 – Review Proc



T. METIN SEZGIN

Review



PROC

⌘ allows to create new procedures

Exp Val = Int + Bool + Proc

Den Val = Int + Bool + Proc

↳ set of values

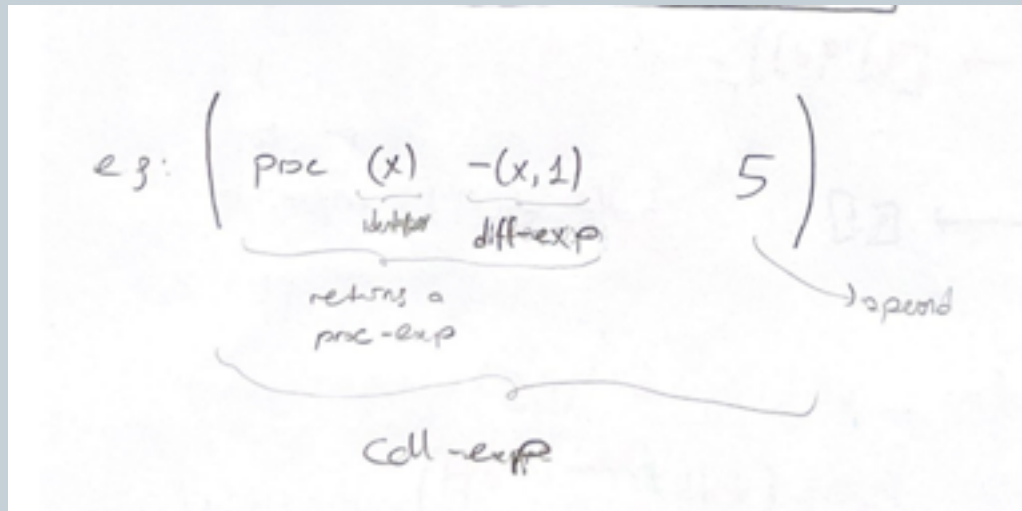
representing procedures

↳ abstract data type

(consider as)

Sude Gungor

Review



Numan Batur

Review



ⓧ Procedure creating and calling

Expression ::= proc (Identifier) Expression

proc-exp (var body) bound variable
or
formal parameter

Expression ::= (Expression Expression)

call-exp (rator rand) value of operand is argument
operand / actual param.
operator

Sude Gungor

Review



Constructor \rightarrow procedure
 ② Why constructor procedure must take 3 args :

$$\hookrightarrow (\text{value-of } (\text{proc-exp } \text{var } \text{body}) \text{ p})$$

$$= (\text{proc-val } (\text{procedure } \text{var } \text{body } \text{p}))$$
 ! constructor \rightarrow like bool-val or num-val (saved-env)
 let x = 200
 in let f = proc (x) - (x, x) — env/p
 in let x = 100
 in let g = proc (x) - (x, x) — done twice
 in - ((f 1), (g 1)) \rightarrow once this is evaluated in environment it is saved env. !

① x bound to 200
 subtracts 200 from its arg. } procedure f

② x bound to 100
 subtracts 100 from its arg. } procedure g

Sude Gungor

Review



- Ⓐ if a value of the operator is `proc-val`,
then apply it to the value of the operand.

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

- Ⓐ What happens when ~~observer~~ ^{apply-procedure} `apply-procedure` is invoked?

- 1 a procedure applied
- 2 body evaluated
- 3 in environment that binds the formal parameter of the procedure to the argument

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

Sude Gungor

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

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 \rightarrow **procedure**

```
(value-of (proc-exp var body)  $\rho$ )  
= (proc-val (procedure var body  $\rho$ ))
```

- Observer \rightarrow **apply-procedure**

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

The intuition behind application



- Extend the environment
- Evaluate the body

```
(apply-procedure (procedure var body  $\rho$ ) val)  
= (value-of body [var=val]  $\rho$ )
```

```

(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)))
```