Lecture 20
Parameter Passing

What is the value of the following expression?

- What happens during evaluation?

```
let p = proc (x) set x = 4
in let a = 3
    in begin (p a); a end
```
Parameter Passing Variations

- Natural (PROC)
- Call-by-value
- Call-by-reference
- Call-by-name (lazy evaluation)
- Call-by-need (lazy evaluation)

let p = proc (x) set x = 4
in let a = 3
   in begin (p a); a end

Evaluates to 3
Call-by-value (IREF)

let p = proc (x) set x = 4
in let a = 3
    in begin (p a); a end

Evaluates to 3

Call-by-reference

let p = proc (x) set x = 4
in let a = 3
    in begin (p a); a end

Evaluates to 4
Another example

let f = proc (x) set x = 44
in let g = proc (y) (f y)
in let z = 55
in begin (g z); z end

CBV → 55
CBR → 44
let f = proc (x) set x = 44
in let g = proc (y) (f y)
in let z = 55
in begin
  (g z);
  z
end

Evaluation trace

entering let g
newref: allocating location 4
entering body of let g with env =
  ([g 4] (f 3) (1 0) (v 1) (x 2))
store =
  ((0 #struct:num-val 1))
  (1 #struct:num-val 5))
  (2 #struct:num-val 10))
  (3 (procedure x ... ([l 0] (v 0) (x 2))))
  (4 (procedure y ... ([f 3] (1 0) (v 1) (x 2))))

entering let z
newref: allocating location 5
entering body of let z with env =
  ([z 5] (g 4) (f 3) (1 0) (v 1) (x 2))
store =
  ((0 #struct:num-val 1))
  (1 #struct:num-val 5))
  (2 #struct:num-val 10))
  (3 (procedure x ... ([l 0] (v 1) (x 2))))
  (4 (procedure y ... ([f 3] (1 0) (v 1) (x 2))))
  (5 #struct:num-val 55))

entering body of proc y with env =
  ([y 5] (f 3) (1 0) (v 1) (x 2))
store =
  ((0 #struct:num-val 1))
  (1 #struct:num-val 5))
  (2 #struct:num-val 10))
  (3 (procedure x ... ([l 0] (v 1) (x 2))))
  (4 (procedure y ... ([f 3] (1 0) (v 1) (x 2))))
  (5 #struct:num-val 55))

entering body of proc x with env =
  ([x 5] (1 0) (v 1) (x 2))
store =
  ((0 #struct:num-val 1))
  (1 #struct:num-val 5))
  (2 #struct:num-val 10))
  (3 (procedure x ... ([l 0] (v 1) (x 2))))
  (4 (procedure y ... ([f 3] (1 0) (v 1) (x 2))))
  (5 #struct:num-val 55))

Uses of call-by-reference

- Multiple return values

let swap = proc (x) proc (y)
let temp = x
in begin
  x = y;
  y = temp
end

in let a = 33
  in let b = 44
    in begin
      (swap a) b);
      (a,b)
    end
Implementing CBR

- Expressed and denoted values remain the same
- Location allocation policy changes
  - If the formal parameter is a variable, pass on the reference
  - Otherwise, put the value of the formal parameter into the memory, pass a reference to it

\[
\begin{align*}
\text{ExpVal} & = \text{Int} + \text{Bool} + \text{Proc} \\
\text{DenVal} & = \text{Ref}(\text{ExpVal})
\end{align*}
\]

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

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

value-of-operand : Exp × Env → Ref
(define value-of-operand
  (lambda (exp env)
    (cases expression exp
      (var-exp var) (apply-env env var))
      (else
        (newref (value-of exp env))))))
```

Another example

```
(let b = 3
 in let p = proc (x) proc(y)
     begin
       set x = 4;
       y
     end
 in ((p b) b)
```

- Here there is variable aliasing
- This evaluates to 4
Lazy evaluation

- Call-by-name
- Call-by-need

```scheme
(inline infinite-loop (x) = infinite-loop(-(x,-1))
in let f = proc (z) 11
   in (f (infinite-loop 0))
```

Thunks

- Save any future work for the future

```scheme
(define-datatype thunk thunk?
(a-thunk
    (exp1 expression?)
    (env environment?)))
```
Implementation (call-by-name)

\[
\begin{align*}
\text{DenVal} & \quad = \quad \text{Ref}(\text{ExpVal} + \text{Thunk}) \\
\text{ExpVal} & \quad = \quad \text{Int} + \text{Bool} + \text{Proc}
\end{align*}
\]

\text{value-of-operand : Exp} \times \text{Env} \rightarrow \text{Ref}

\begin{lambda} \text{exp} \text{env} \end{lambda}

\begin{cases}
\text{var-exp (var) (apply-env env var)} \\
\text{else} \\
\text{(newref (a-thunk exp env))}
\end{cases}

\text{value-of-thunk : Thunk} \rightarrow \text{ExpVal}

\begin{lambda} \text{th} \end{lambda}

\begin{cases}
\text{cases thunk th} \\
\text{(a-thunk (expl saved-env) \text{(value-of expl saved-env))}}
\end{cases}

Memoization (call-by-need)

\text{var-exp (var)}

\begin{lambda} \text{ref1 (apply-env env var)} \end{lambda}

\begin{Cases}
\text{let ((w (deref ref1)))} \\
\text{if \text{(expval? w)}} \\
\text{w} \\
\text{(value-of-thunk w))}
\end{Cases}

\text{let ((vall (value-of-thunk w))}
\begin{lambda}
\text{begin}
\text{(setref! ref1 vall)
\text{vall))}
\end{lambda}