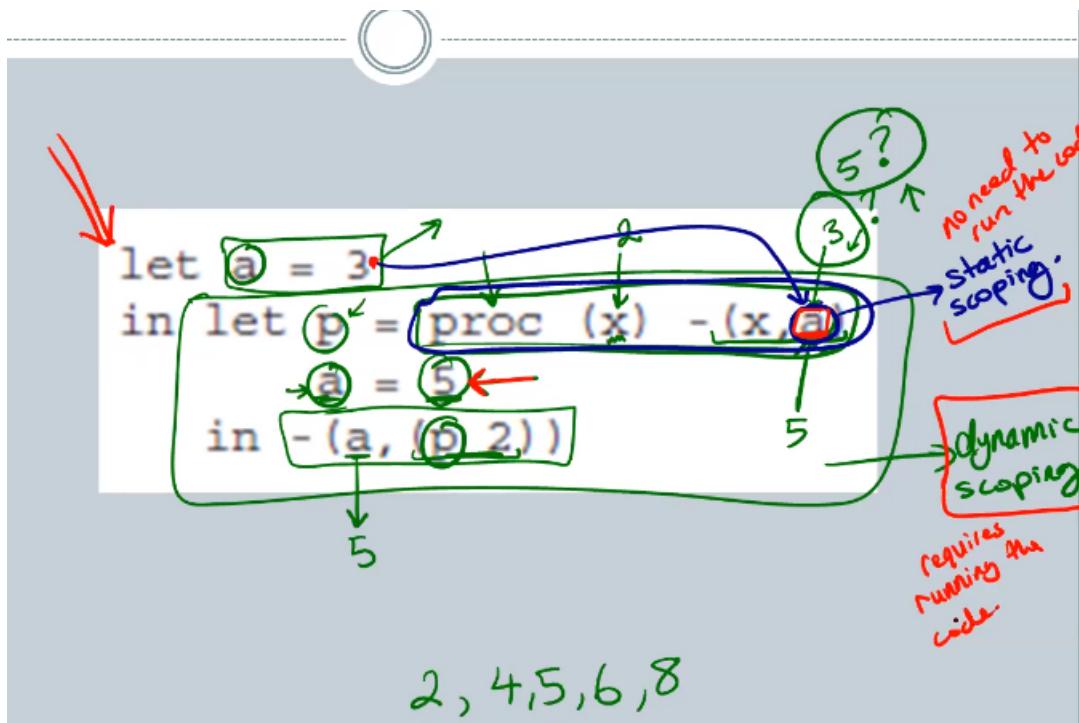
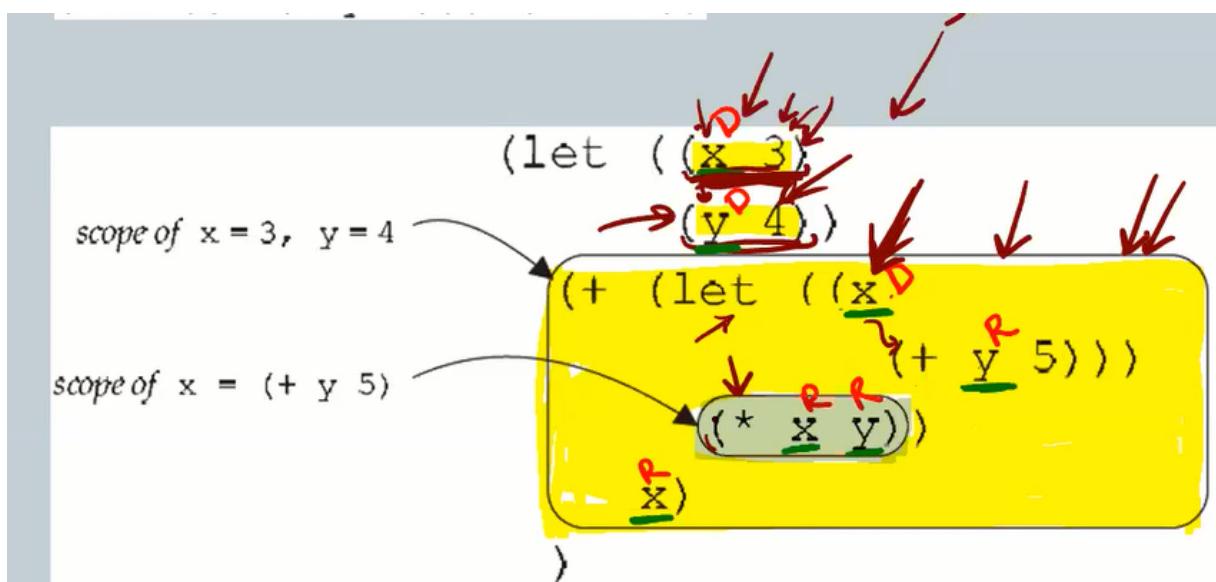


Lecture 15



Static scoping takes the previous declared value above its level

Dynamic scoping takes the last declared value of a



$\rightarrow (\text{let } ((\underline{x} \ 3) \\ \quad (\underline{y} \ 4)) \\ (+ \ (\text{let } ((\underline{x} \\ \quad (\underline{* \ x \ y})) \\ \quad (\underline{x1}) \) \)$

Call this $\frac{x_1}{y_1}$

Call this x₂.

Here x refers to x2

Here x refers to x_1

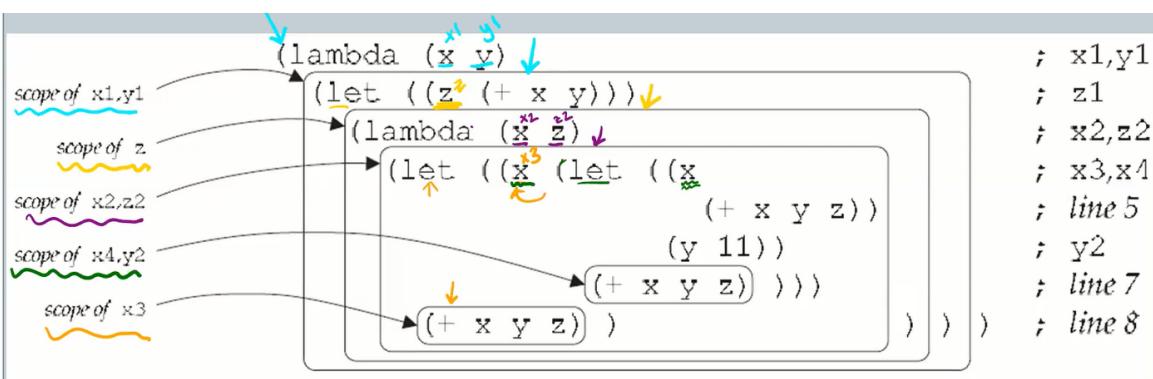
```
(let ((x 3)
      (y 4))
  (+ (let ((x
            (+ y 5)))
       (* x y))
     x))
```

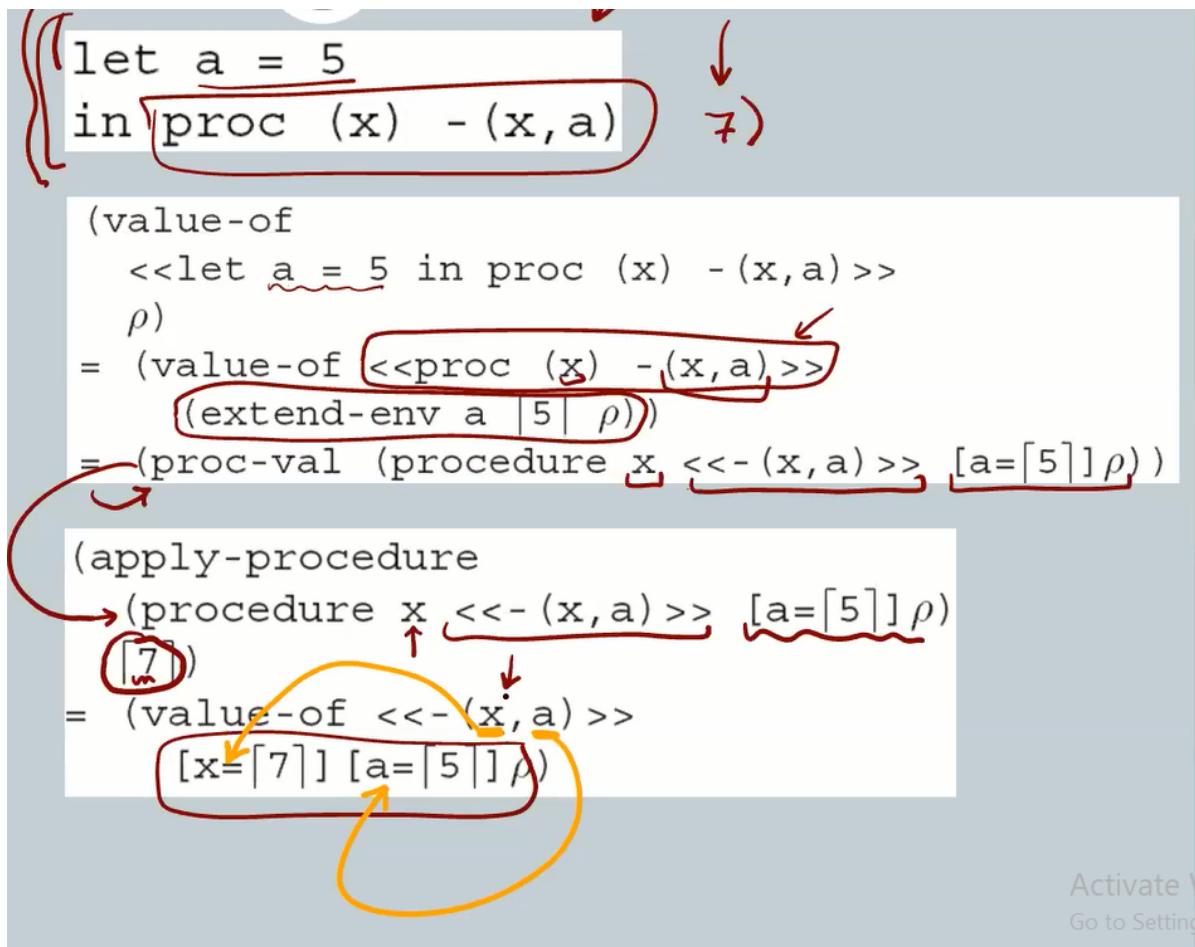
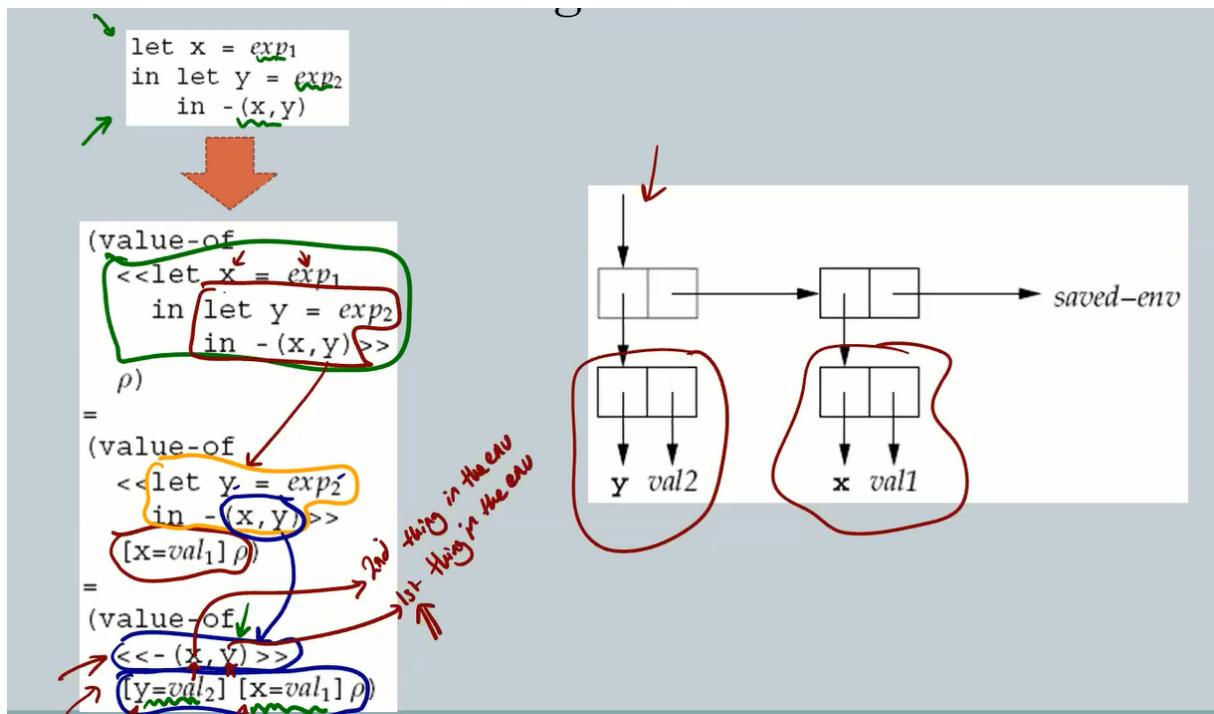
Call this x_1

Call this x2

Here x refers to x2

Here x refers to x_1

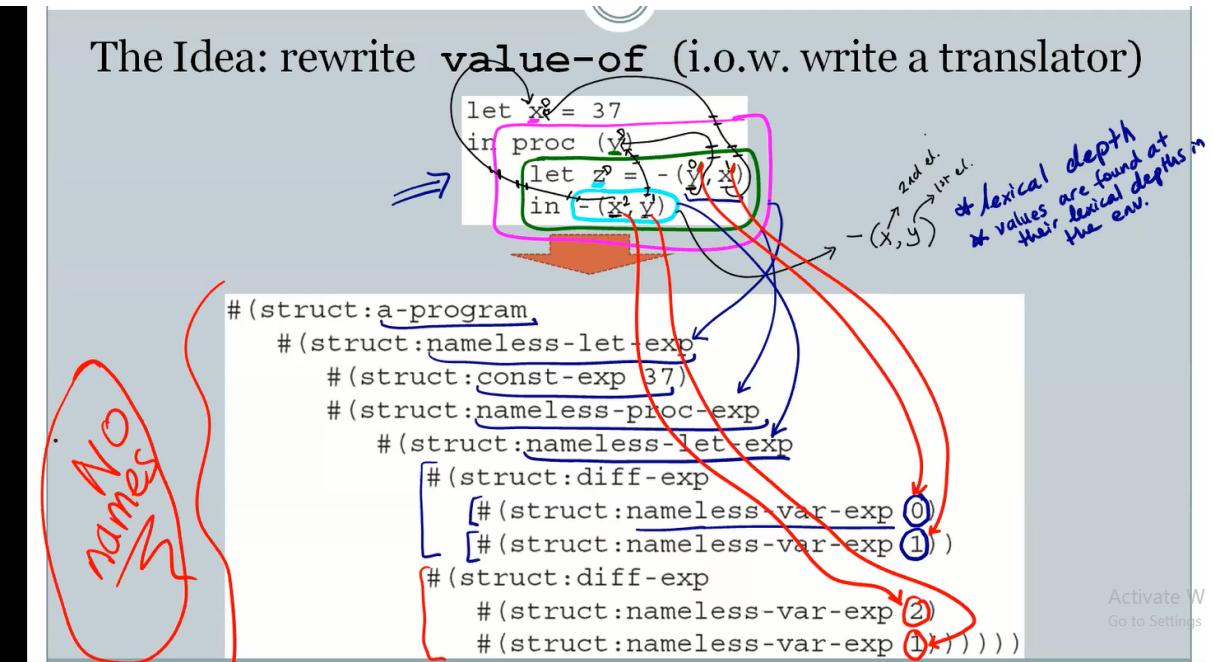




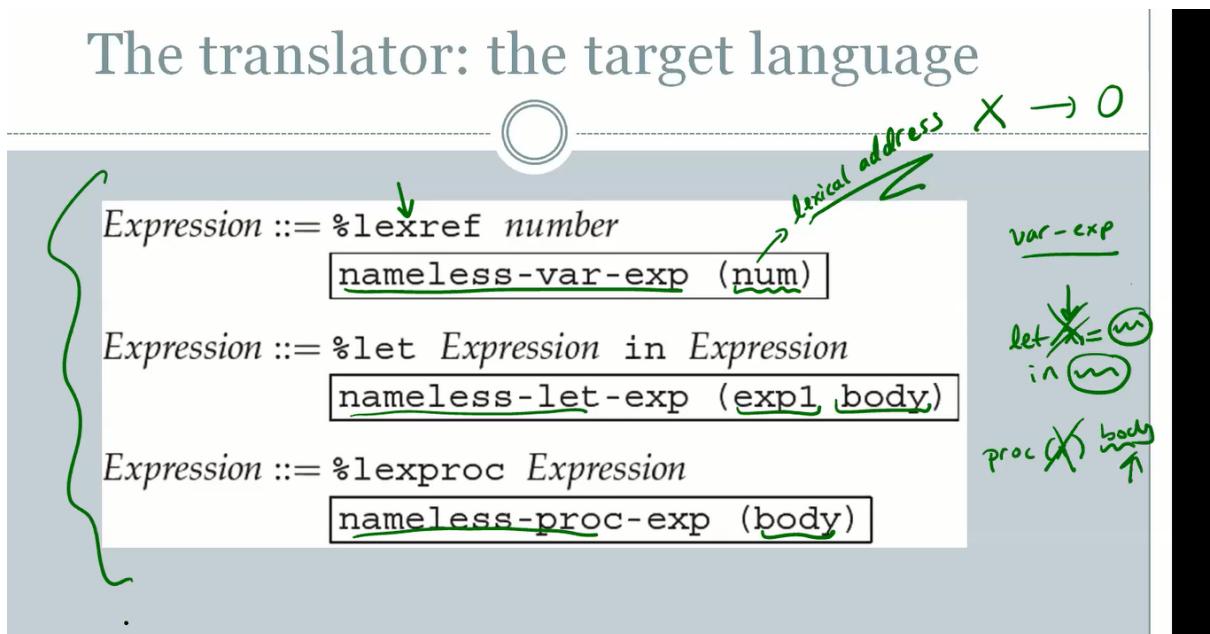
Activate W
Go to Settings

LEXICAL DEPTH:

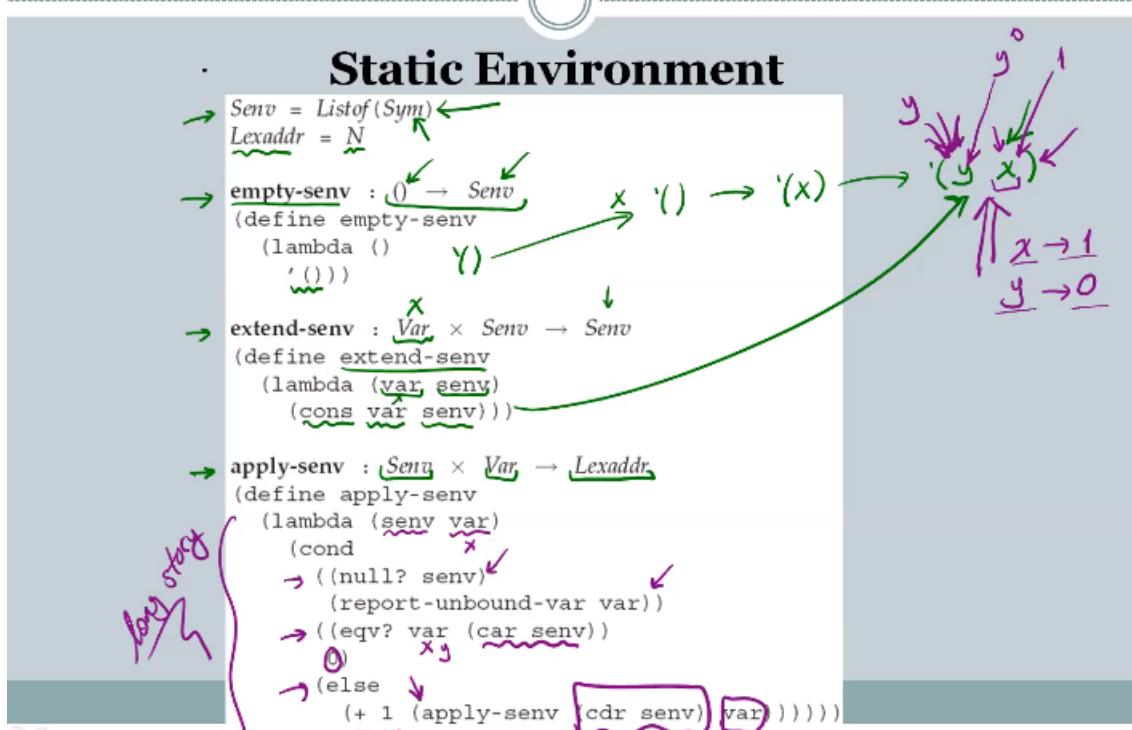
The Idea: rewrite **value-of** (i.o.w. write a translator)



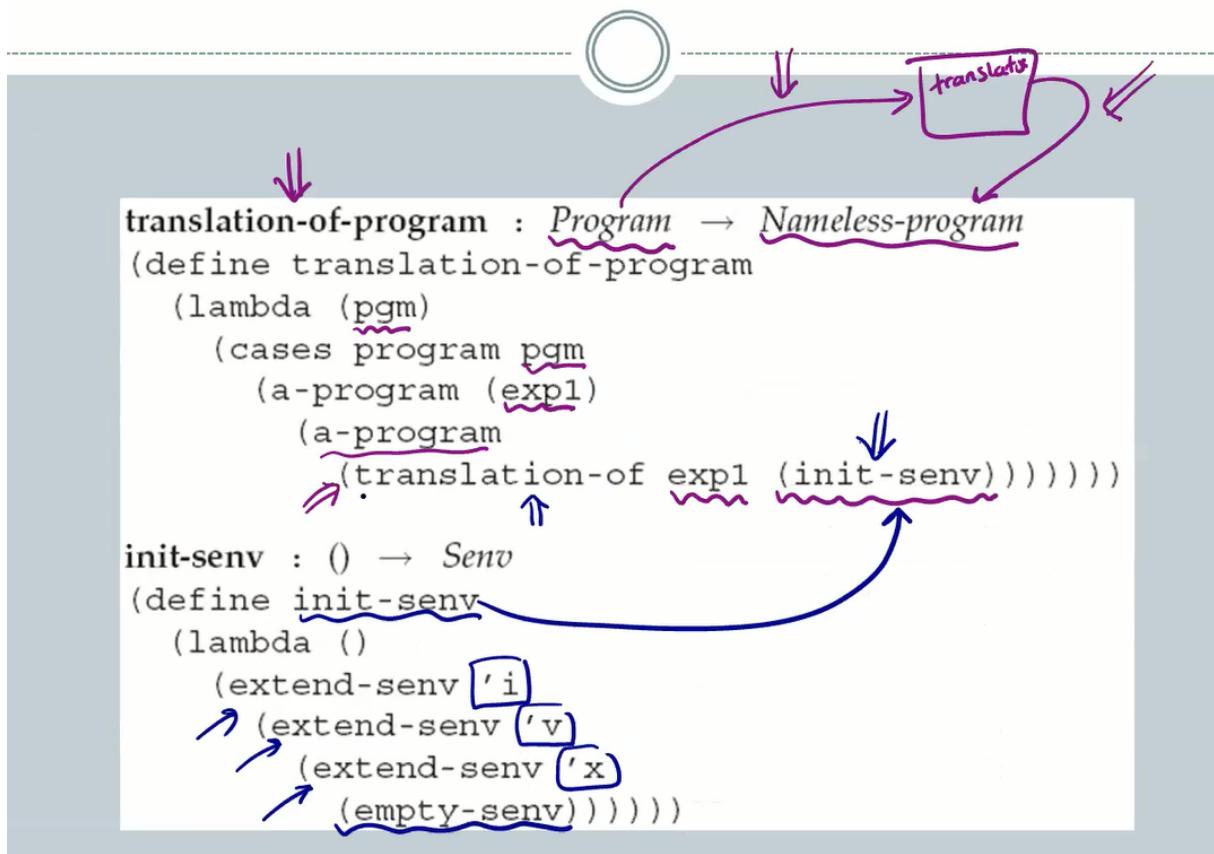
The translator: the target language

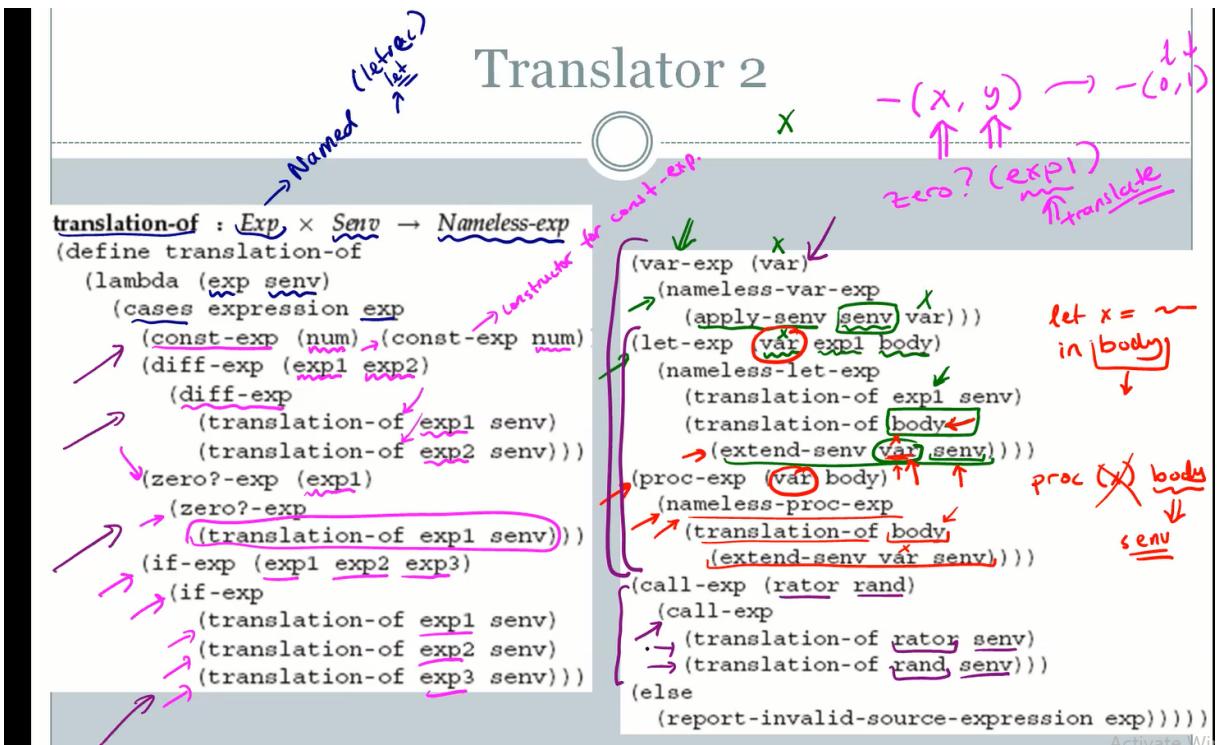


The translator: $\text{Exp} \times \text{Senv} \rightarrow \text{NamelessExp}$

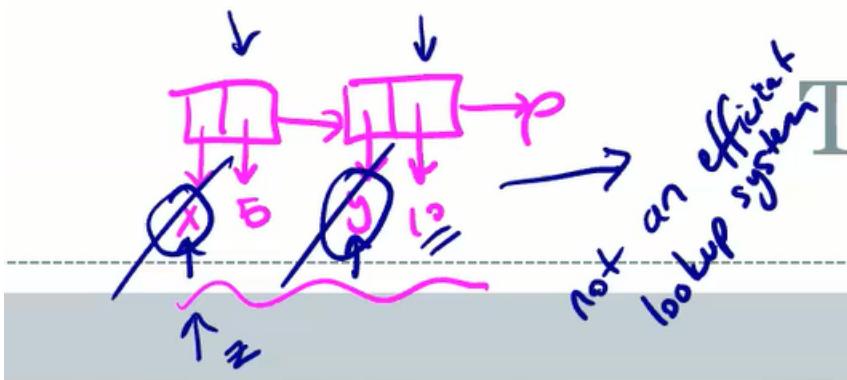


Translator 1



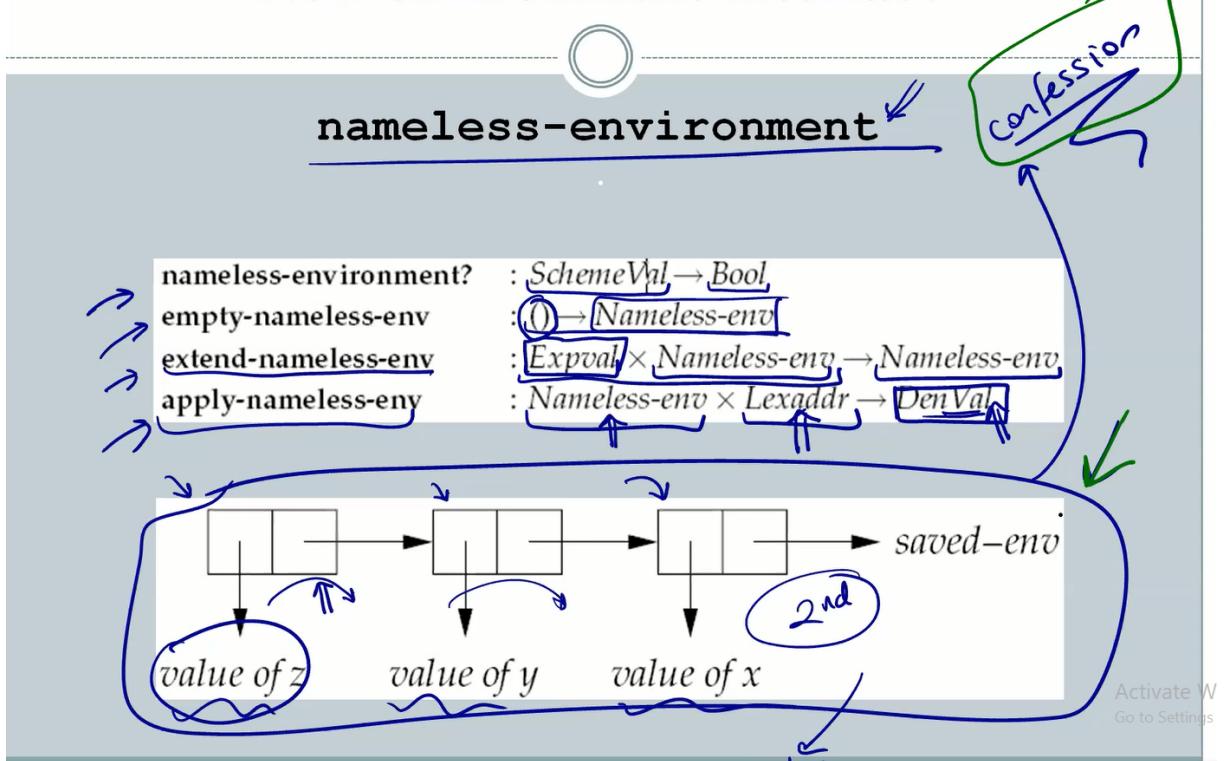


BY MAKING THE PROGRAM NAMELESS WE MAKE THE INTERPRETATION OR EXECUTION OF THE PROGRAM SIMPLER SINCE INSTEAD OF LOOKING FOR A ONE BY ONE THROUGH THE LEVELS INSIDE A PROGRAM BELOW LIKE A LINKED LIST WE DIRECTLY GO TO THE ADDRESS

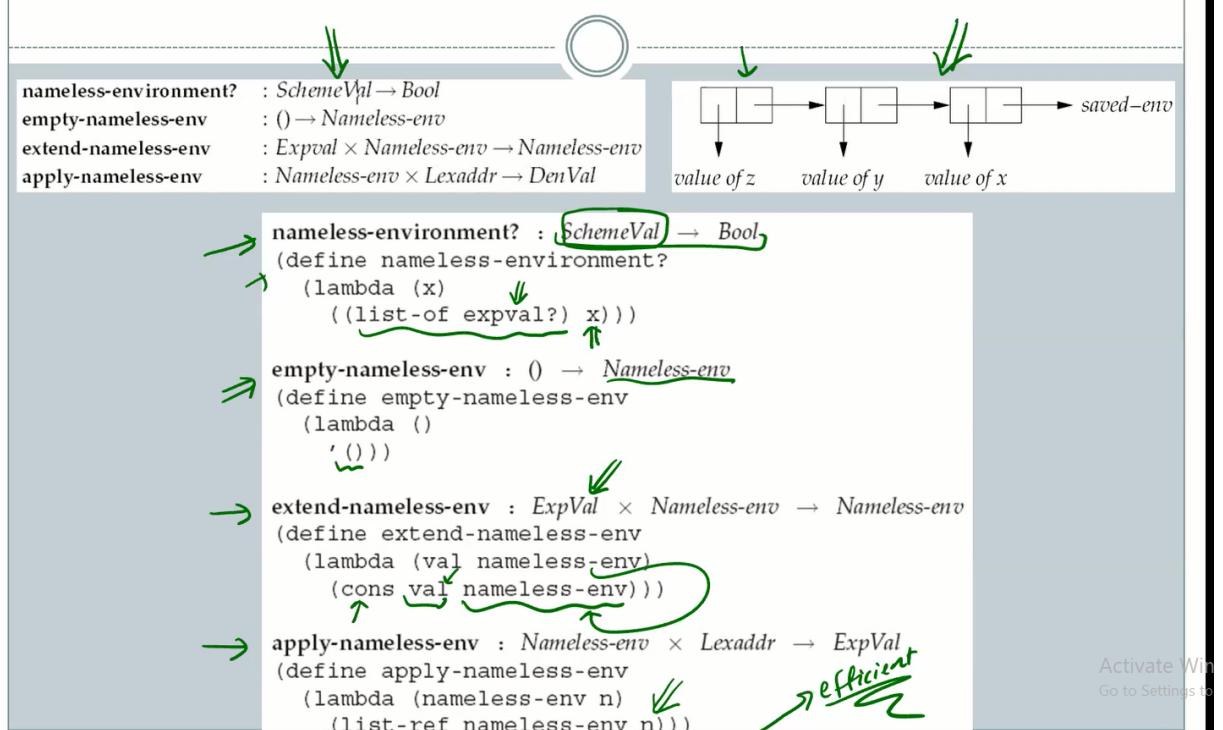


THE TRANSLATOR TRANSLATES IT TO A NAMELESS VERSION AND FEEDS IT TO VALUE WHICH IS THE INTERPRETER SO NOW WE NEED TO HAVE A NEW INTERPRETER WHERE IT TAKES IN NAMELESS

New environment interface

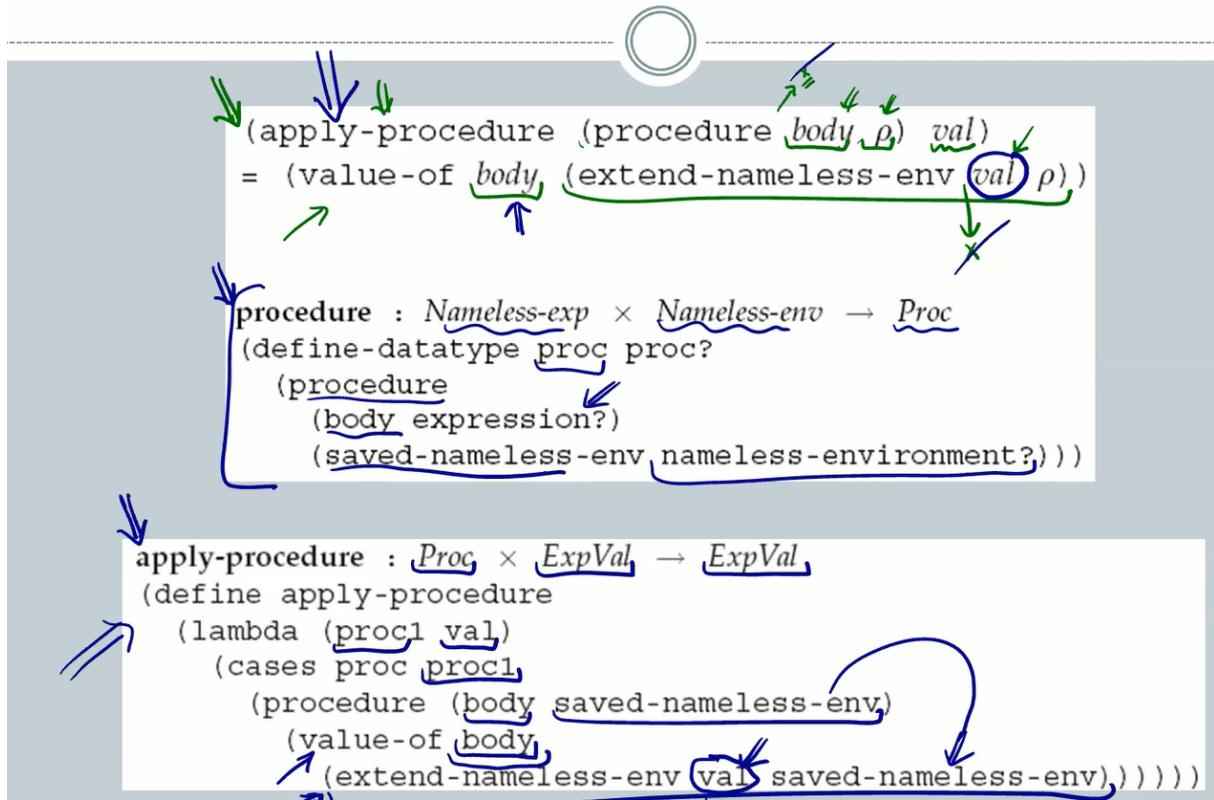


New environment interface



WHAT IS THIS:

Procedure specification and implementation



LOOK AGAIN AT HOW PROC IS APPLIED

VALUEOF:

NAMELESS-VAR-EXP: TAKES THE NAMELESSENV AND A VARIABLE THIS WAS SOMETHING LIKE X BEFORE BUT NOW IT IS THE ADDRESS AND THEN RETURNS THE VALUE AT THE ADDRESS OR IN OTHER WORDS VALUE OF X WHICH IN THIS CASE IS 7.

FOR LET THE IDEA IS TO FEED THE BODY TO THE VALUE OF WHERE WE ADD ANOTHER PARAMATER LIKE X = 12 CHECK THE SLIDE WITH ;ECTION 12- REVIEW LET THIS TIME WE SIMPLY ADD 12 AND THAT IS IT:

Interpreter for the new language

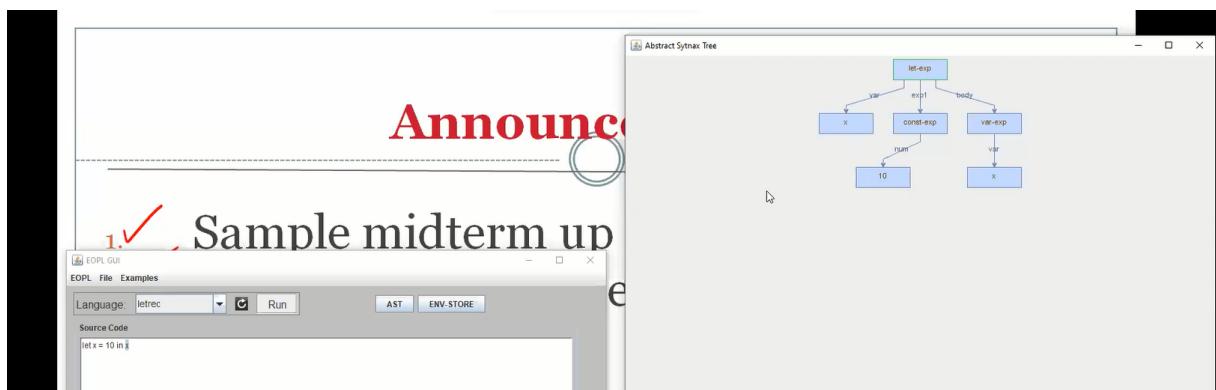
Tam ekranдан çıkmak için Esc tuşuna basın

```

value-of : Nameless-exp × Nameless-env → ExpVal
(define value-of
  (lambda (exp nameless-env)
    (cases expression exp
      ((const-exp (num) ...as before...))
      ((diff-exp exp1 exp2) ...as before...))
      ((zero?-exp (exp1) ...as before...))
      ((if-exp (exp1) (exp2) (exp3)) ...as before...))
      ((call-exp (rator rand) ...as before...))
      ((nameless-var-exp (n))
       (apply-nameless-env nameless-env n)))
      (else
        (report-invalid-translated-expression exp))))
```

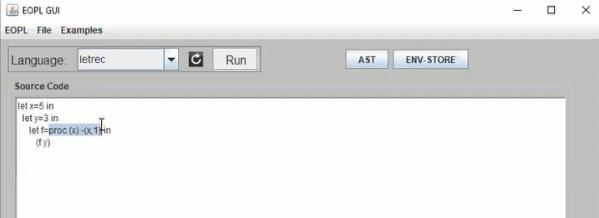
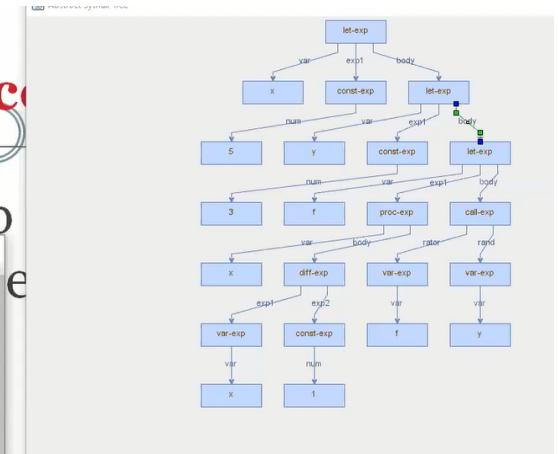
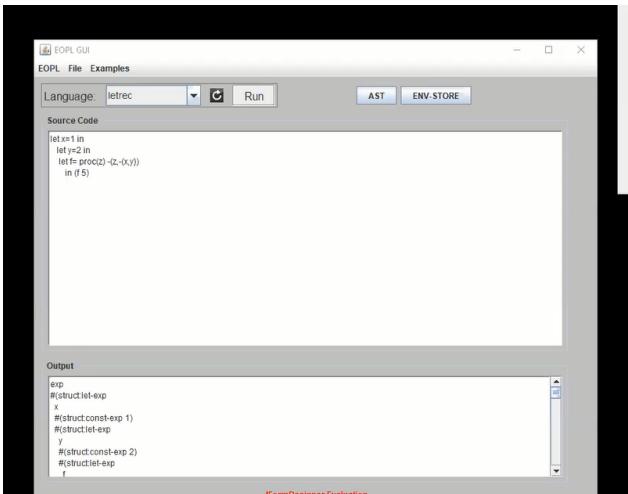
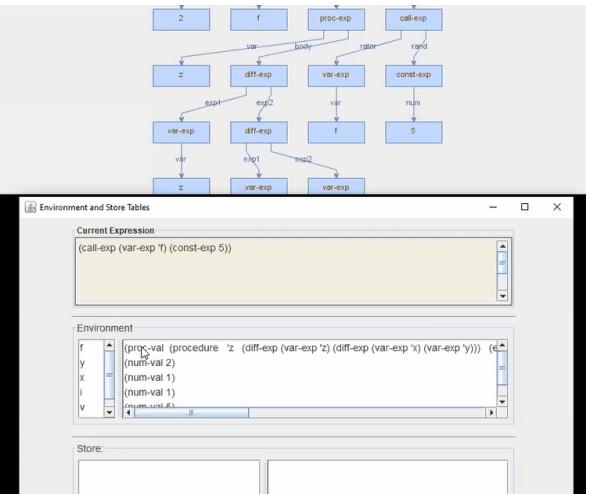
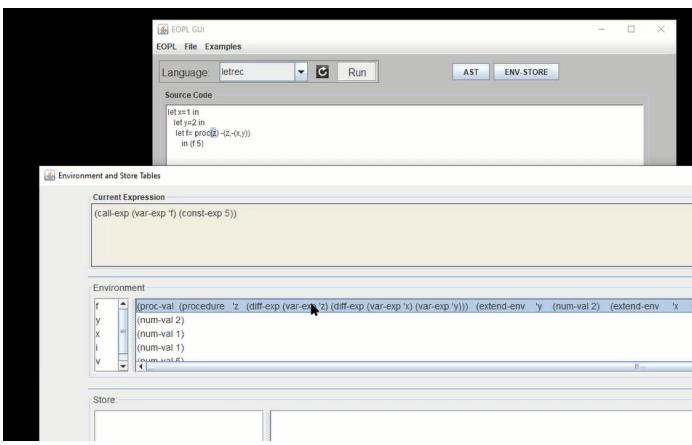
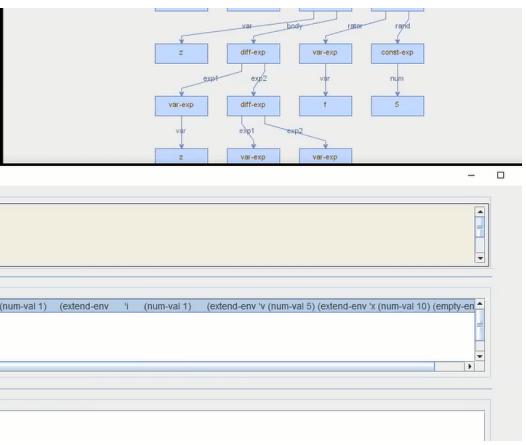
Annotations:

- Red arrows point from the first five cases to a large pink circle labeled "X".
- A blue arrow points from the "nameless-var-exp" case to a circled "7" with the handwritten note "Lexical address of the var.".
- A blue arrow points from the "nameless-var-exp" case to the "apply-nameless-env" call.
- A blue arrow points from the "nameless-var-exp" case to the "else" clause.
- A blue arrow points from the "nameless-var-exp" case to the "report-invalid-translated-expression" call.
- A blue arrow points from the "nameless-let-exp" case to the "value-of" call in its body.
- A blue arrow points from the "nameless-let-exp" case to the "extend-nameless-env" call in its body.
- A blue arrow points from the "nameless-let-exp" case to the "else" clause.
- A blue arrow points from the "nameless-let-exp" case to the "report-invalid-translated-expression" call in its body.
- A blue arrow points from the "nameless-proc-exp" case to the "proc-val" field.
- A blue arrow points from the "nameless-proc-exp" case to the "procedure" call in its body.
- A blue arrow points from the "nameless-proc-exp" case to the "else" clause.
- A blue arrow points from the "nameless-proc-exp" case to the "report-invalid-translated-expression" call in its body.



Announce

✓ Sample midterm up

Language: letrec

Source Code

```
(letrec f(x) = if zero?(x) then 0 else -( (f(-(x,1)), -2) in (f 4)
```

Output

```
#(struct:num-val 5)
 #(struct:extend-env x #(struct:num-val 10) #(struct:empty-env))))))

(num-val 8)
```

Source Code

```
(letrec f(x) = if zero?(x) then 0 else -( (f(-(x,1)), -2) in (f 4)
```

Output

```
#(struct:num-val 5)
 #(struct:extend-env x #(struct:num-val 10) #(struct:empty-env))))))

(num-val 8)
```

Environment and Store Tables

Current Expression

```
(var-exp 'f')
```

Environment

f	(if-exp (zero?-exp (var-exp 'x)) (const-exp 0) (diff-exp (call-exp (var-exp 'f)) (diff-exp (var-exp 'x) (const-exp 1))) (const-exp -2)))
i	(num-val 1)
v	(num-val 5)
x	(num-val 10)

Source Code

```
letrec f(x) = if zero?(x) then 0 else -(f(-x,1)), -2 in (f 4)
```

Output

```
#(struct:num-val 5)
#(struct:extend-env x #(struct:num-val 10) #(struct:empty-env)))))

(num-val 8)
```

Environment and Store Tables

Current Expression

```
(zero?-exp (var-exp 'x))
(const-exp 0)
(diff-exp
  (call-exp (var-exp 'f) (diff-exp (var-exp 'x) (const-exp 1)))
  (const-exp -2)))
```

Environment

x	(num-val 4)
f	(if-exp (zero?-exp (var-exp 'x)) (const-exp 0) (diff-exp (call-exp (var-exp 'f) (diff-exp (var-exp 'x) (const-exp 1))) (const-exp -2)))
i	(num-val 1)
v	(num-val 5)
x	(num-val 10)

EREF VS IREF : U CAN STORE INFO

C VS JAVA POINTER STORAGE

- Denotable and Expressed values

$$\begin{aligned} \underline{\text{ExpVal}} &= \underline{\text{Int}} + \underline{\text{Bool}} + \underline{\text{Proc}} + \boxed{\underline{\text{Ref}}(\underline{\text{ExpVal}})} \\ \underline{\text{DenVal}} &= \underline{\text{ExpVal}} \end{aligned}$$

```
Welcome to DrRacket, version 6.1.1 [3m].
Language: racket; memory limit: 128 MB.
> (let ((fact (lambda (n)
  (if (zero? n)
    1
    (* n (fact (sub1 n)))))))
  (fact 4))
 $\times$  fact: undefined;
cannot reference an identifier before its definition
> (letrec ((fact (lambda (n)
  (if (zero? n)
    1
    (* n (fact (sub1 n)))))))
  (fact 4))
 $\times$  letrec: bad syntax (missing body) in: (letrec ((fact (lambda (n) (if (zero? n) 1 (* n (fact (sub1 n)))))))
  (fact 5))
120
>
```

The new design

- Denotable and Expressed values

$$\begin{aligned} \underline{\text{ExpVal}} &= \boxed{\text{Int} + \text{Bool} + \text{Proc}} + \boxed{\text{Ref}(\text{ExpVal})} \\ \underline{\text{DenVal}} &= \underline{\text{ExpVal}} \end{aligned}$$

- Three new operations

- newref
- deref
- setref

Let $x = \text{NEWREF}(0)$: set x equal to a location where its value = 0

$\text{DREF}(x)$: return the value located at the address = x

$\text{SETREF}(x, 13)$: set the value located at address x to 13

Example: references help us share variables

Tam ekrandan çıkmak için Esc tuşuna basın

```
let x = newref(0)
in letrec even(dummy)
    = if zero?(deref(x))
      then 1
      else begin
            setref(x, -(deref(x), 1));
            (odd 888)
          end
odd(dummy)
= if zero?(deref(x))
  then 0
  else begin
        setref(x, -(deref(x), 1));
        (even 888)
      end
in begin setref(x, 13); (odd 888) end
```

X= even will return if Even() called 1

X= even will return if odd() called 0

X= odd will return if Even() called 0

X= odd will return if odd() called 1

Example: references help us create hidden state

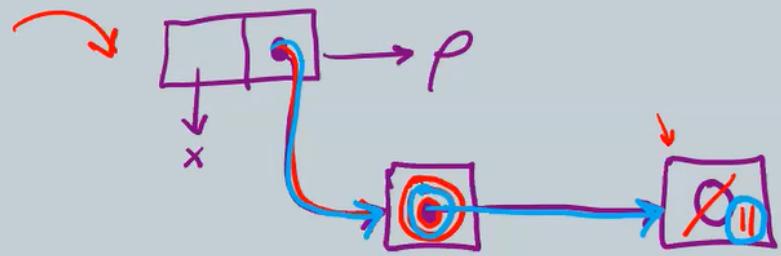
```
let g = let counter = newref(0)
      in proc(dummy)
         begin
            setref(counter, -(deref(counter), -1));
            deref(counter)
         end
      in let a = (g.11)
         in let b = (g.11)
            in -(a,b)
```

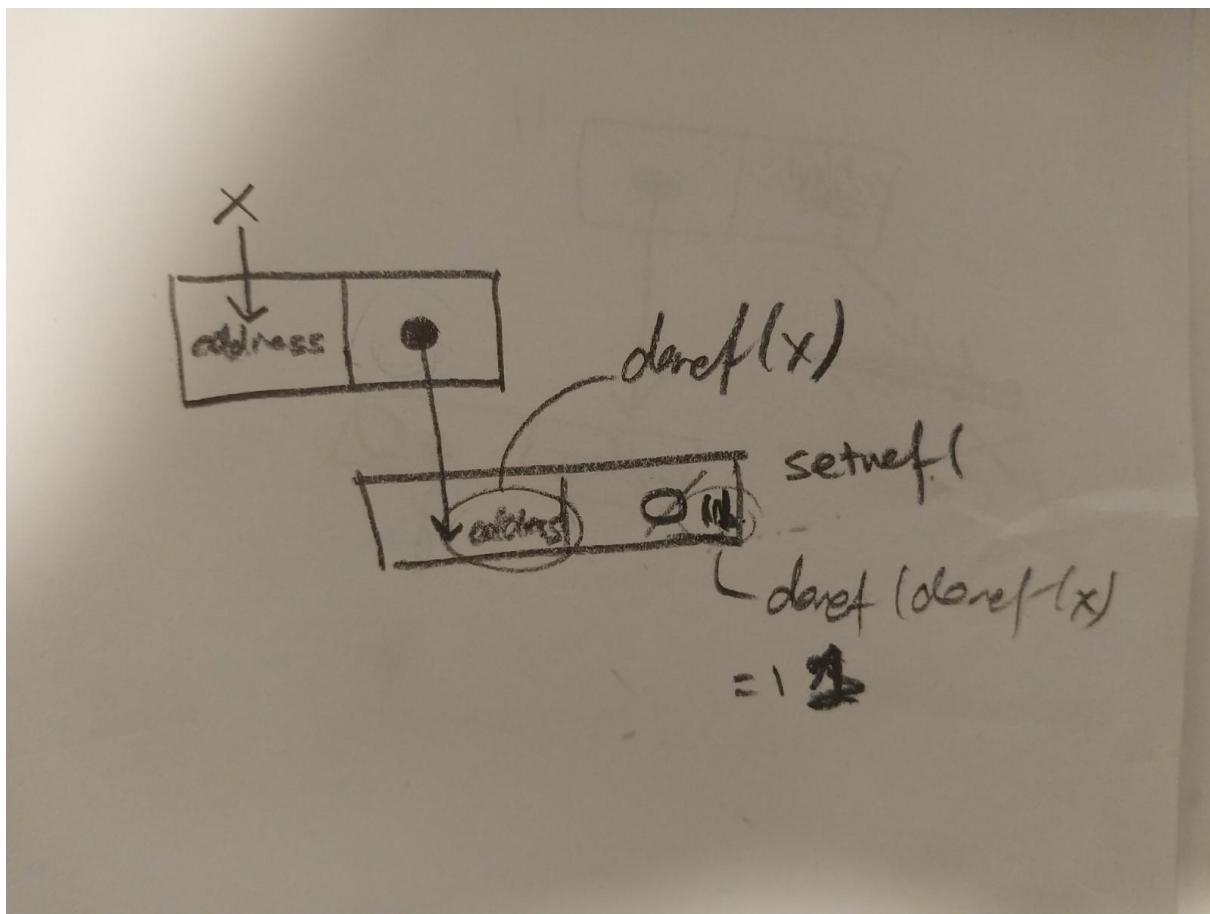
The entire expression evaluates to -1

Metin Sezai: Kind of relate that to this right so I'm gonna

Activat
Go to Set

```
let x = newref(newref(0))  
in begin  
    setref(deref(x), 11);  
    deref(deref(x))  
end
```





Store passing specifications

- The new **value-of** $(\text{value-of } exp_1 \rho \sigma_0) = (val_1, \sigma_1)$
- Example $(\text{value-of } (\text{const-exp } n) \rho \sigma) = (n, \sigma)$
- More examples
 - $(\text{value-of } exp_1 \rho \sigma_0) = (val_1, \sigma_1)$
 - $(\text{value-of } exp_2 \rho \sigma_1) = (val_2, \sigma_2)$
 - $(\text{value-of } (\text{diff-exp } exp_1 exp_2) \rho \sigma_D) = ([val_1 - val_2], \sigma_D)$
 - $(\text{value-of } (exp_1 \rho \sigma_0) = (val_1, \sigma_1)$
 - $\begin{cases} (\text{value-of } (\text{if-exp } exp_1 exp_2 exp_3) \rho \sigma_0) \\ = \begin{cases} (\text{value-of } exp_2 \rho \sigma_1) & \text{if } (\text{expval} \rightarrow \text{bool } val_1) = \#t \\ (\text{value-of } exp_3 \rho \sigma_1) & \text{if } (\text{expval} \rightarrow \text{bool } val_1) = \#f \end{cases} \end{cases}$

Grammar specification

- The new grammar

```

Expression ::= newref (Expression)
             | newref-exp (exp1)

Expression ::= deref (Expression)
             | deref-exp (exp1)

Expression ::= setref (Expression , Expression)
             | setref-exp (exp1 exp2)
  
```

- Specification

\checkmark $\frac{(\text{value-of } \exp) \rho \sigma_0 = (l, \sigma_1) \quad l \notin \text{dom}(\sigma_1)}{(\text{value-of } (\text{newref-exp } \exp) \rho \sigma_0) = ((\text{ref-val } l), [l=val] \sigma_1)}$	$l \notin \text{dom}(\sigma_1)$ $[l=val] \sigma_1$ <i>a new address</i>
\checkmark $\frac{(\text{value-of } \exp) \rho \sigma_0 = (l, \sigma_1)}{(\text{value-of } (\text{deref-exp } \exp) \rho \sigma_0) = (\sigma_1(l), \sigma_1)}$	<i>return the contents of location l</i>
\checkmark $\frac{(\text{value-of } \exp_1) \rho \sigma_0 = (l, \sigma_1) \quad (\text{value-of } \exp_2) \rho \sigma_1 = (val, \sigma_2)}{(\text{value-of } (\text{setref-exp } \exp_1 \exp_2) \rho \sigma_0) = ([23], [l=val] \sigma_2)}$	(l, σ_1) (val, σ_2) $[23]$ $[l=val] \sigma_2$

-(5,2)
3
Activate Window
Go to Settings

value-of-program : *Program* → *ExpVal*

```
(define value-of-program
  (lambda (pgm)
    "..."
    → (initialize-store!)
      (cases program pgm
        (a-program (expl)
          (value-of expl (init-env)))))))
```

$\prod \quad \prod$

Implementation of Stores

empty-store : () → *Sto*

```
(define empty-store
  (lambda () '()))
```

get-store : () → *Sto*

```
(define get-store
  (lambda () the-store))
```

reference? : *SchemeVal* → *Bool*

```
(define reference?
  (lambda (v)
    (integer? v)))
```

deref : *Ref* → *ExpVal*

```
(define deref
  (lambda (ref)
    (list-ref the-store ref)))
```

usage: A Scheme variable containing the current state of the store. Initially set to a dummy value.

```
(define the-store 'uninitialized)
```

initialize-store! : () → *Unspecified*

```
(define initialize-store!
  (lambda ()
    (set! the-store (empty-store))))
```

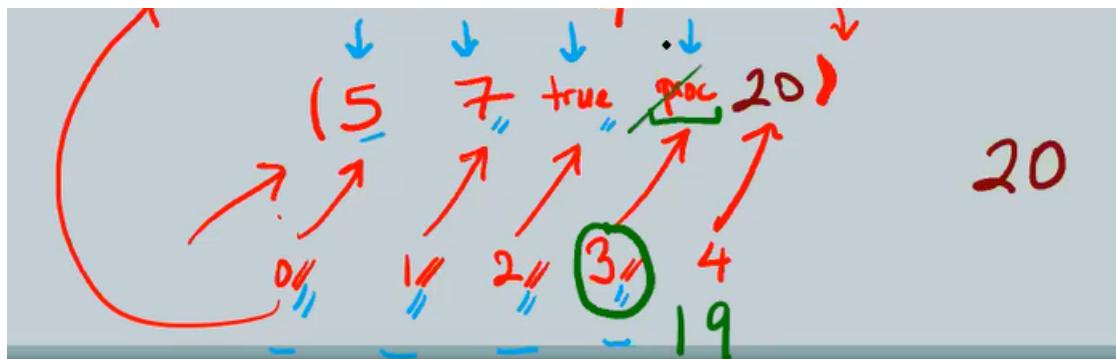
newref : *ExpVal* → *Ref*

```
(define newref
  (lambda (val)
    (let ((next-ref (length the-store)))
      → (set! the-store (append the-store (list val)))
```

↑ (20) ↑

(5) ↓ 7 ↓ true ↓ 20 ↓ 4 ↓ 3 ↓ 4 ↓ 20

Implementation of setref:



Lets say you want to set the value at location 3 to 19, the idea is to when u call setref it one by one iterates thru locations until it gets to 3, so 5 7 true and when it gets to that location it puts the 19 and cons the remainder or rest of the list which in this case is 20.

setref!

```

setref! : Ref × ExpVal → Unspecified
usage: sets the-store to a state like the original, but with
position ref containing val.
(define setref!
  → (lambda (ref val)
      (set! the-store
            (letrec
              ((setref-inner
                usage: returns a list like storel, except that
                position refl contains val.
                (lambda (storel refl)
                  (cond
                    ((null? storel)
                     → (report-invalid-reference ref the-store))
                    ((zero? refl)
                     → (cons val (cdr storel)))
                    (else
                     (cons
                       (car storel)
                       → (setref-inner
                           (cdr storel) (- refl 1))))))))
              (setref-inner the-store ref))))))

Activat
Go to Sel
  
```

The code shows the implementation of the `setref!` function. It uses a helper function `setref-inner` to modify the store. The `setref-inner` function takes a store and a reference index as arguments. If the index is zero, it conses the value onto the store. Otherwise, it conses the first element of the store onto the result of a recursive call with the rest of the store and the decremented index. Handwritten annotations explain the process: a blue bracket on the left indicates the range of elements being modified, and green annotations show the step-by-step construction of the new list [5, 9, 3, true] from the original [5, 7, true, 20]. A blue arrow points from the original list to the new list, and a green arrow points from the original list to the final mutated value 19.

Implementation

`newref-exp`, `deref-exp`, `setref-exp`

Activate
Go to Settings

Implicit references

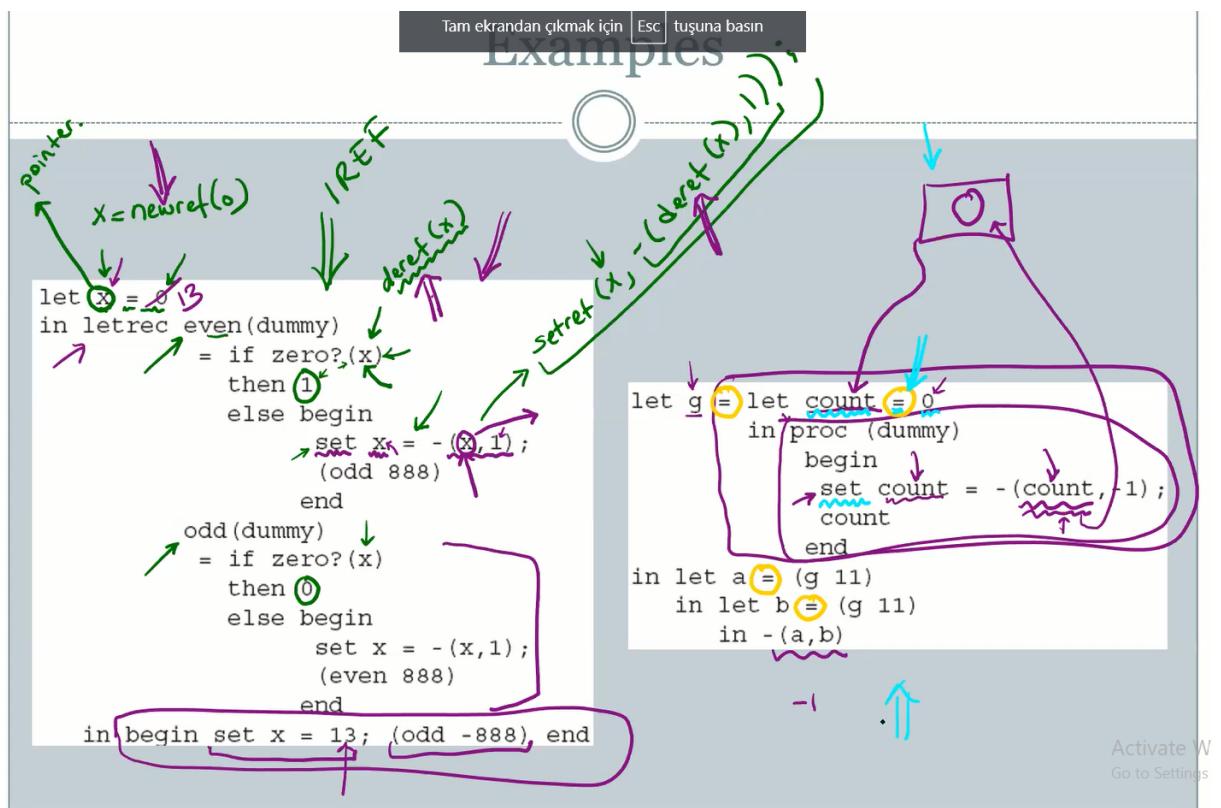
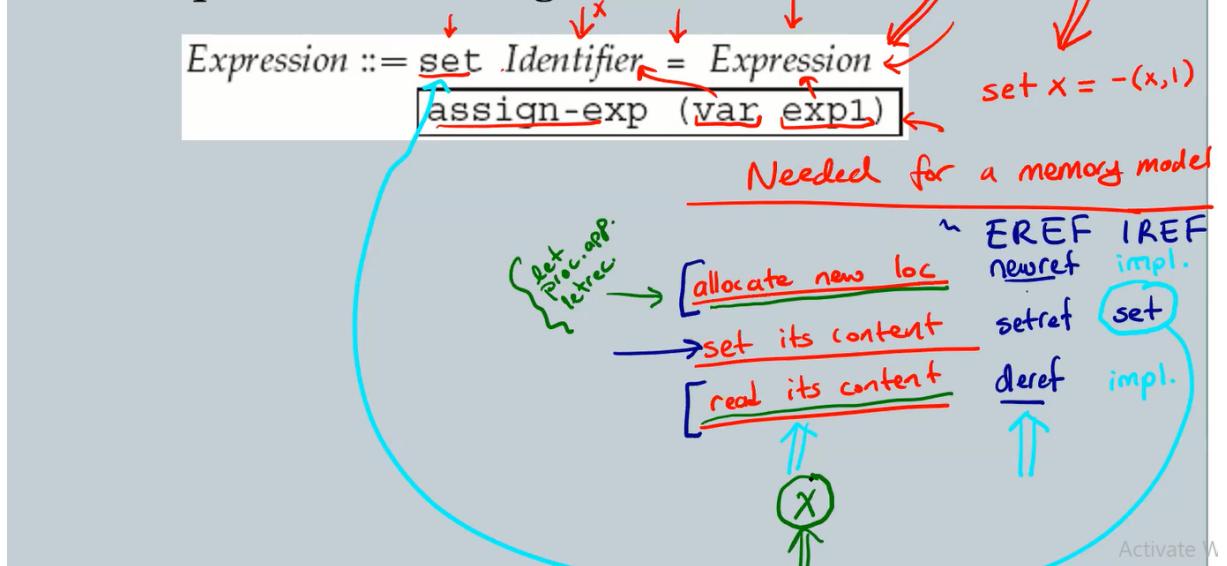
- IREF
 - ✓ ○ References are instantiated by the interpreter
 - ✓ ○ All denoted values are references to expressed values
 - Each binding operation introduces a location
 - Let
 - letrec
 - proc
 - Pointers to stores are saved in the environment

Activate Wi
Go to Settings

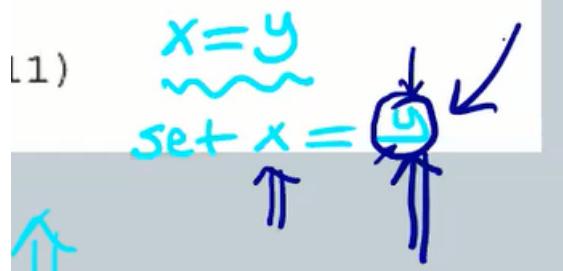
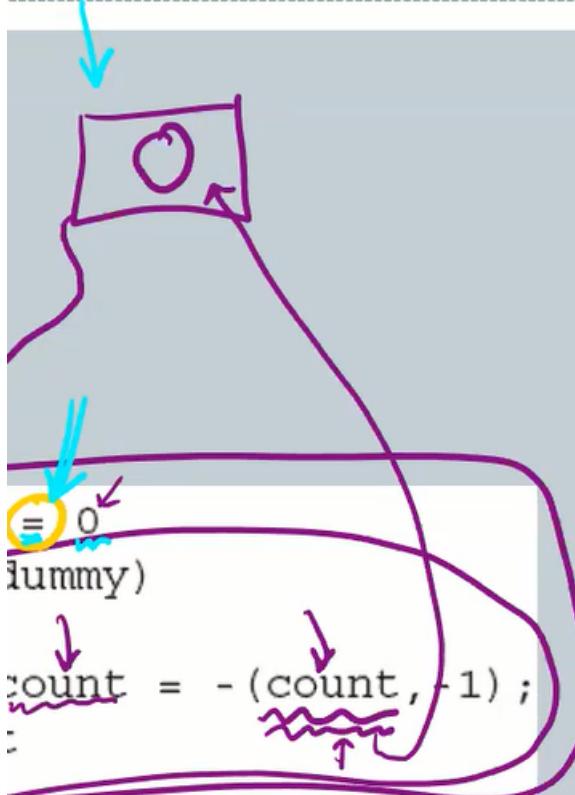
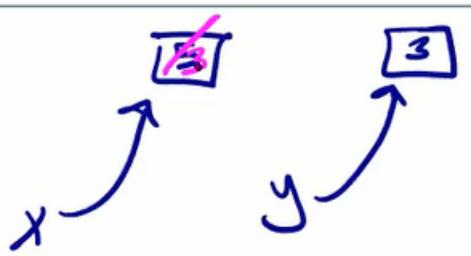
$$\begin{array}{rcl} \text{ExpVal} & = & \text{Int} + \text{Bool} + \text{Proc} \\ \text{DenVal} & = & \boxed{\text{Ref(ExpVal)}} \end{array}$$

New grammar ↗

- A set operation for assignment



IREF cannot make references to references unlike EREF



Behavior specification

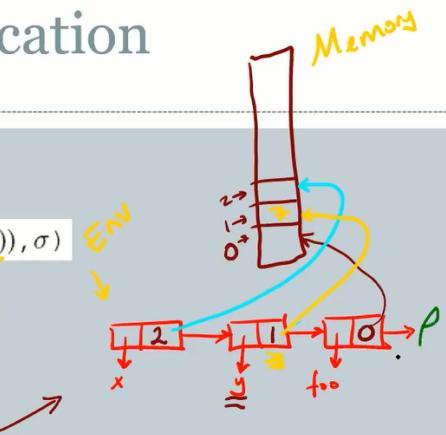
- var-exp

$$(\text{value-of } (\text{var-exp } \text{var}) \rho \sigma) = (\sigma(\rho(\text{var})), \sigma)$$

y

$$= \sigma(1)$$

= 7



let foo = ~ in
let y = 7 in
let x = ~ in
y

Activat

Behavior specification

- var-exp

$$(\text{value-of } (\text{var-exp } \text{var}) \rho \sigma) = (\sigma(\rho(\text{var})), \sigma)$$

y

$$\sigma(\rho(y)) = \sigma(1) = 7$$

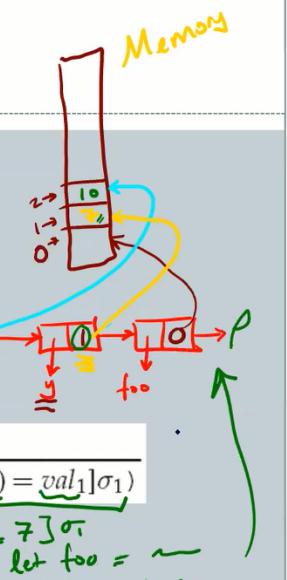
- assign-exp

$$(\text{value-of } \text{exp}_1 \rho \sigma_0) = (\underline{\text{val}_1}, \sigma_1)$$

$$(\text{value-of } (\text{assign-exp } \text{var } \text{exp}_1) \rho \sigma_0) = ([27], [\rho(\text{var}) = \underline{\text{val}_1}] \sigma_1)$$

x
y

- apply-procedure



$\rho(x) = 7$ σ_1
let foo = ~
let y = 1 in
let x = 2 in
set x = y

Activate W
Go to Settings

Behavior specification

var-exp

$$\text{value-of}(\text{var-exp } \text{var}) \rho \sigma = (\sigma(\rho(\text{var})), \sigma)$$

ENV: $\boxed{1 \ 2} \rightarrow \boxed{0 \ 1} \rightarrow \boxed{+ \ 0}$
MEM: $(\boxed{20 \ 30 \ 70})$

let $f = \text{proc}$ in
let $y = 20$ in
let $z = 30$ in
 $(f \ \boxed{70})$

assign-exp

$$\frac{\text{value-of } \text{exp}_1 \rho \sigma_0 = (\text{val}_1, \sigma_1)}{\text{value-of}(\text{assign-exp } \text{var } \text{exp}_1) \rho \sigma_0 = ([27], [\rho(\text{var}) = \text{val}_1] \sigma_1)}$$

set $x = 70$

apply-procedure

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

Implementation

- **var-exp**

$\text{value-of}(\text{var-exp } \text{var}) \rightarrow (\text{deref } (\text{apply-env env var}))$

address

- **assign-exp**

$\text{assign-exp}(\text{var } \text{exp}_1)$

begin
 setref!
 $(\text{apply-env env var})$
 $(\text{value-of exp}_1 \text{env})$
 $(\text{num-val } 27))$

set $x = 19$

- **apply-procedure**

$\text{apply-procedure} : \text{Proc} \times \text{ExpVal} \rightarrow \text{ExpVal}$

```

(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body)
        (extend-env var (newref val) saved-env)))))))
  
```

let $f = \text{proc}(x) 5$
in $(f 2)$

Implementation

Reference instantiations

let $x = 1$
in $(x, 2)$

✓ apply-procedure

✓ let

✓ letrec

```

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 (newref val) saved-env))))))
  → (let-exp (var exp1 body)
    (let ((val1 (value-of exp1 env))
      (value-of body)
      (extend-env var (newref val1) env)))
    (extend-env-rec (p-names b-vars p-bodies saved-env)
      (let ((n (location-search-var p-names)))
        (if n
          (newref
            (proc-val
              (procedure
                (list-ref b-vars n)
                (list-ref p-bodies n)
                env)))
          (apply-env saved-env search-var))))))

REF
  
```

In addition we want mutation

- New grammar

<code>✓ newpair</code> : <u>ExpVal × ExpVal → MutPair</u> <code>✓ left</code> : <u>MutPair → ExpVal</u> <code>✓ right</code> : <u>MutPair → ExpVal</u> <code>setleft</code> : <u>MutPair × ExpVal → Unspecified</u> <code>setright</code> : <u>MutPair × ExpVal → Unspecified</u>	<i>cons</i> <i>set x = 5</i>
---	---------------------------------

- New set of

- Denotables
- Expressibles

```

(define-datatype expval expval?
  (num-val
    (value number?))
  (bool-val
    (boolean boolean?))
  (proc-val
    (proc proc?))
  (mutpair-val
    (p mutpair?)))
  
```

$\begin{cases} \text{ExpVal} &= \text{Int} + \text{Bool} + \text{Proc} + \text{MutPair} \\ \text{DenVal} &= \text{Ref}(\text{ExpVal}) \\ \text{MutPair} &= \text{Ref}(\text{ExpVal}) \times \text{Ref}(\text{ExpVal}) \end{cases}$	<i>set x = 5</i>
--	------------------

Activate Wi
Go to Settings

New scheme functions for pair management

```

make-pair : ExpVal × ExpVal → MutPair
(define make-pair
  (lambda (val1 val2)
    (a-pair
      (newref val1)
      (newref val2)))))

left : MutPair → ExpVal
(define left
  (lambda (p)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (deref left-loc))))))

right : MutPair → ExpVal
(define right
  (lambda (p)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (deref right-loc))))))

setleft : MutPair × ExpVal → Unspecified
(define setleft
  (lambda (p val)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (setref! left-loc val))))))

setright : MutPair × ExpVal → Unspecified
(define setright
  (lambda (p val)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (setref! right-loc val)))))


$$(\text{val} \cdot \text{val}) \quad (\text{make-pair } \text{val} \cdot \text{val})$$


```

Interpreter = valueof

The Interpreter

(value-of (exp env)
(cases (exp)

```

(newpair-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (mutpair-val (make-pair val1 val2)))))

(left-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((p1 (expval->mutpair val1)))
      (left p1)))))

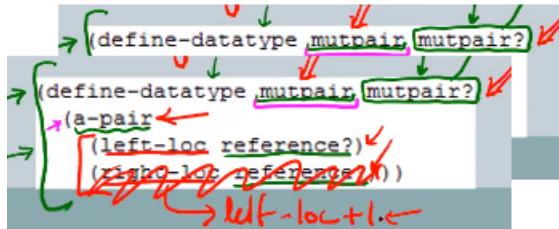
(right-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((p1 (expval->mutpair val1)))
      (right p1)))))

(setleft-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (let ((p (expval->mutpair val1)))
      (begin
        (setleft p val2)
        (num-val 82)))))

(setright-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (let ((p (expval->mutpair val1)))
      (begin
        (setright p val2)
        (num-val 83)))))

newpair
left
right
setleft
setright

```



A different representation for mutable pairs

```

mutpair? : SchemeVal → Bool
(define mutpair?
  (lambda (v)
    (reference? v)))

make-pair : ExpVal × ExpVal → MutPair
(define make-pair
  (lambda (val1 val2)
    (let ((ref1 (newref val1)))
      (let ((ref2 (newref val2)))
        (ref1)))))

left : MutPair → ExpVal
(define left
  (lambda (p)
    (deref p)))

right : MutPair → ExpVal
(define right
  (lambda (p)
    (deref (+ 1 p)))))

setleft : MutPair × ExpVal → Unspecified
(define setleft
  (lambda (p val)
    (setref! p val)))

setright : MutPair × ExpVal → Unspecified
(define setright
  (lambda (p val)
    (setref! (+ 1 p) val)))

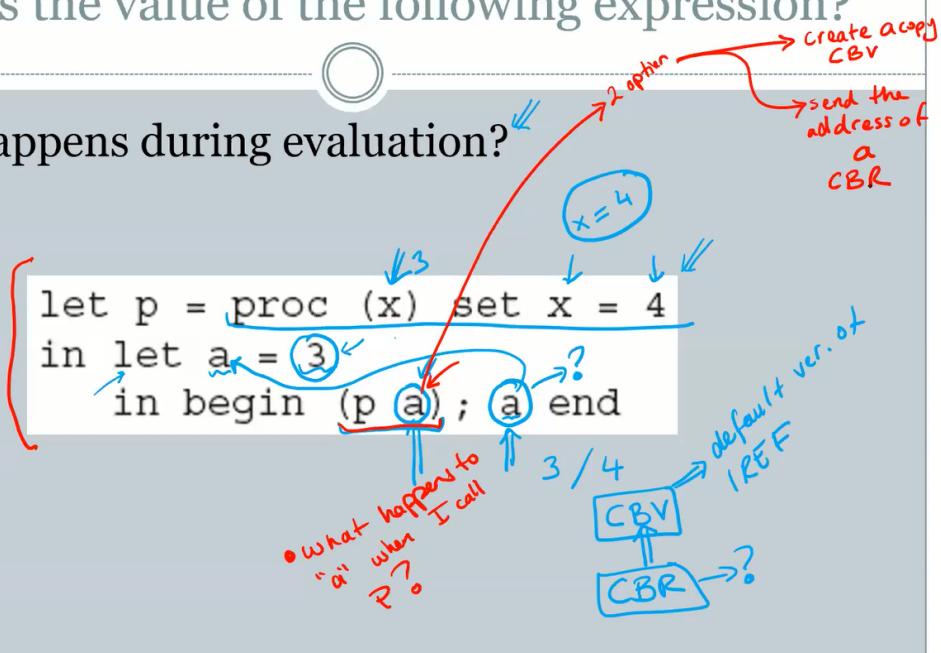
```

The screenshot shows a Scheme editor with handwritten annotations. Red annotations include:
- A red arrow pointing to the `mutpair?` definition.
- Red boxes around the `(reference? v)` call in the `mutpair?` definition.
- Red boxes around the `(newref val1)` and `(newref val2)` calls in the `make-pair` definition.
- Red boxes around the `(deref p)` call in the `left` function.
- Red boxes around the `(deref (+ 1 p))` call in the `right` function.
- Red boxes around the `(setref! p val)` and `(setref! (+ 1 p) val)` calls in the `setleft` and `setright` functions respectively.
- A red circle with a question mark is at the top left.
- A blue circle with a question mark is at the top right.
- A large blue curly brace on the right side groups the `right`, `setleft`, and `setright` definitions.
- Blue annotations include:
- A blue box around the `(+ 1 p)` in the `right` definition.
- Blue boxes around the `(p val)` and `(+ 1 p) val` in the `setleft` and `setright` definitions.
- Blue numbers 19, 20, 21, 22, 23, 24 are scattered around the right side of the code area.
- A blue arrow points from the `(deref p)` in the `left` function to the `(deref (+ 1 p))` in the `right` function.
- A blue arrow points from the `(setref! p val)` in the `setleft` function to the `(setref! (+ 1 p) val)` in the `setright` function.
- A blue arrow points from the `(+ 1 p)` in the `setright` function back to the `(+ 1 p)` in the `right` function.
- A blue box highlights the `Activate` button at the bottom right.

- Alright, so how would you, for example, implement stacks. I would implement stacks by keeping
40:07
- Metin Sezgin: Track of the pointers that we would need in order to be able to push and pop things off the stack and the access to top value. If I'm implementing an array, how would I implement an array. I probably
40:20
- Metin Sezgin: Maybe store the beginning of the array in the memory, assuming that an array is really a sequence of contiguous
40:36
- Metin Sezgin: memory locations. Maybe I'll just save the pointer to the beginning of the array and maybe I'll also save, how many items I have in the array

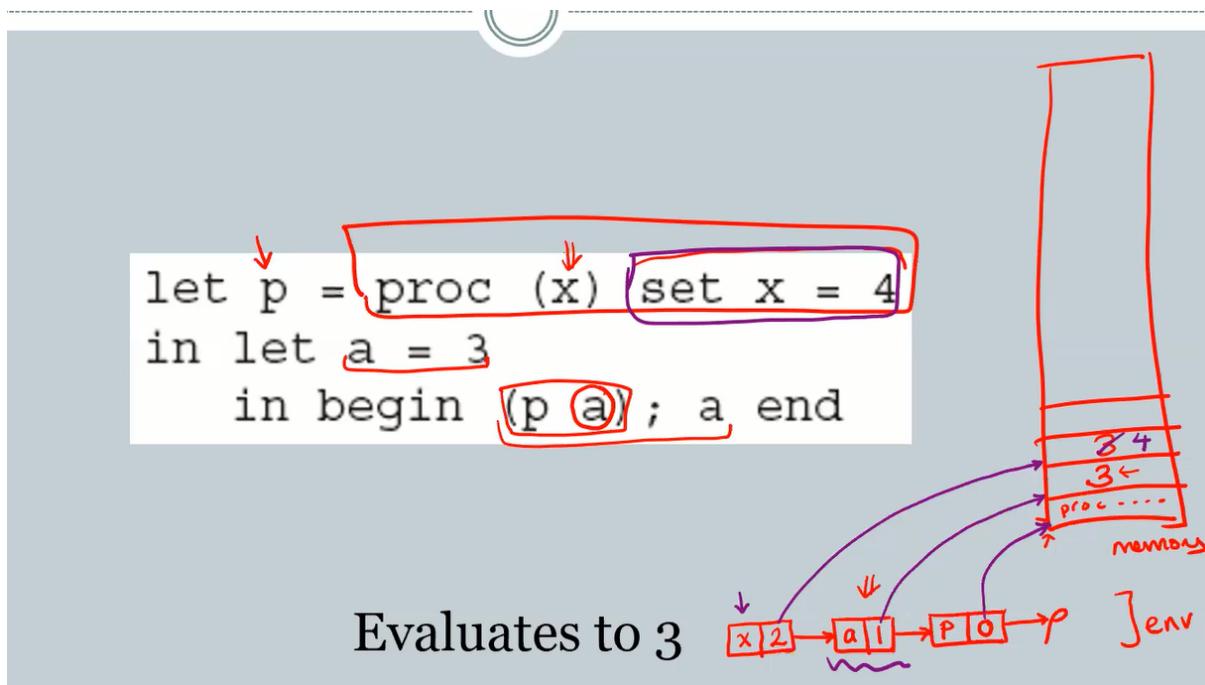
What is the value of the following expression?

- What happens during evaluation?

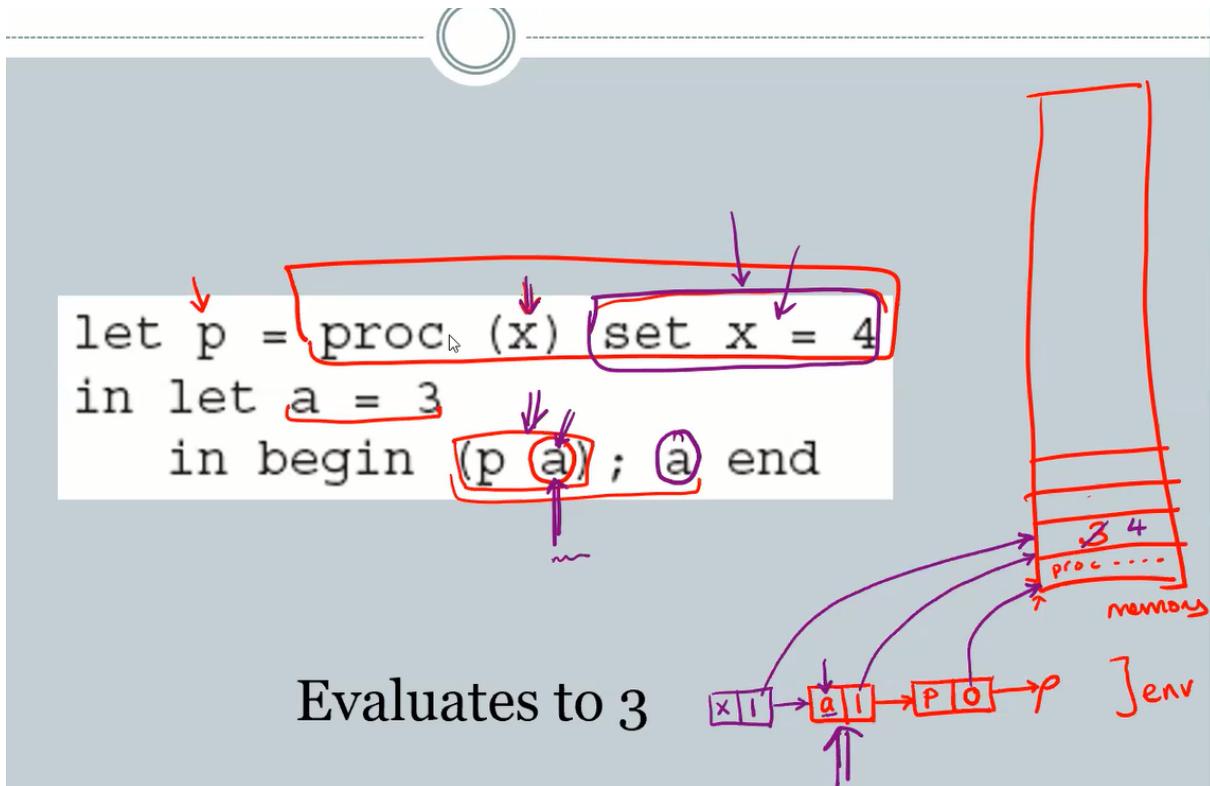


because default of iref is cbv or call by value we end up with a 3 output

IREF IN CBV:

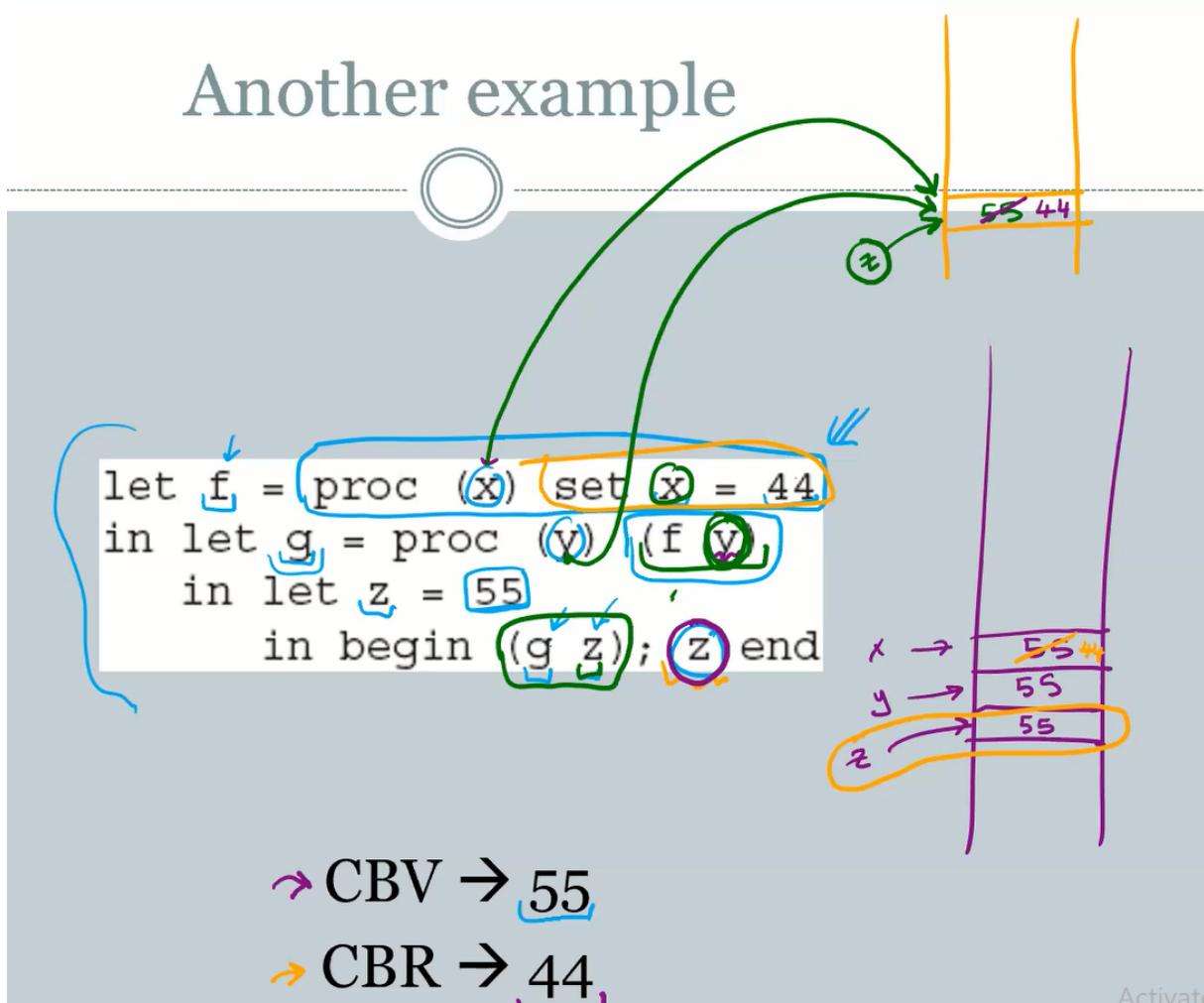


IREF IN CBR:



CBR: Swapping numbers with a function

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

```

Evaluation trace

```

> (run "
let f = proc (x) set x = 44
in let g = proc (y) (f y)
in let z = 55
in begin
  (g z);
  z
end")
newref: allocating location 0
newref: allocating location 1
newref: allocating location 2
entering let f
newref: allocating location 3
entering body of let f with env =
((f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2)))))

newref: allocating location 4
newref: allocating location 5
newref: allocating location 6
entering let g
newref: allocating location 7
entering body of let g with env =
((g 4) (f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2)))))

newref: allocating location 8
newref: allocating location 9
newref: allocating location 10
entering let z
newref: allocating location 11
entering body of let z with env =
((z 5) (g 4) (f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
 (5 #(struct:num-val 55)))

```

```

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

```

Activate

```

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) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 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) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
 (5 #(struct:num-val 55)))

```

```

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

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

```

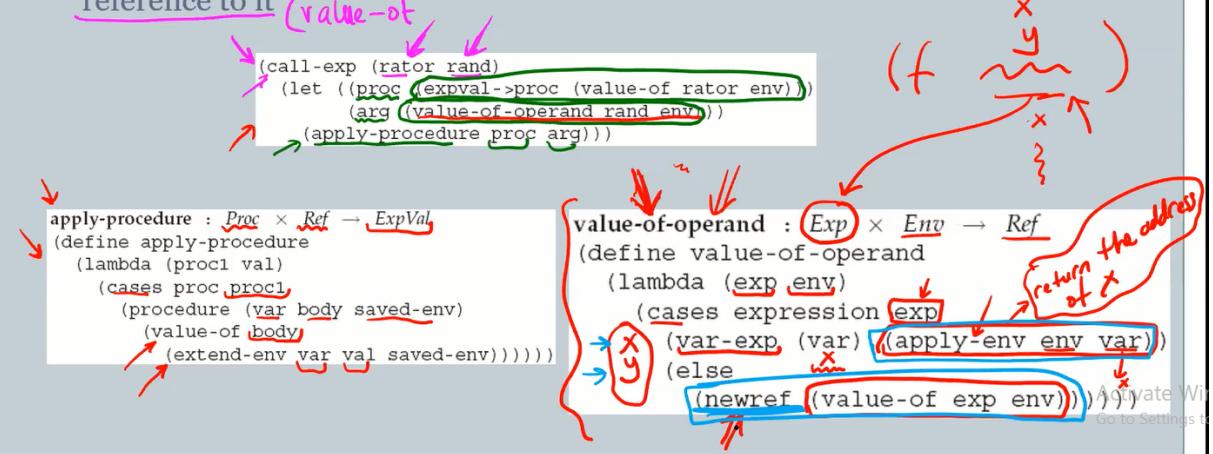
Activate

- bit got added Y equals five. Right. So why the input parameter is pointing at this location five, which above is CBR case

Implementing CBR

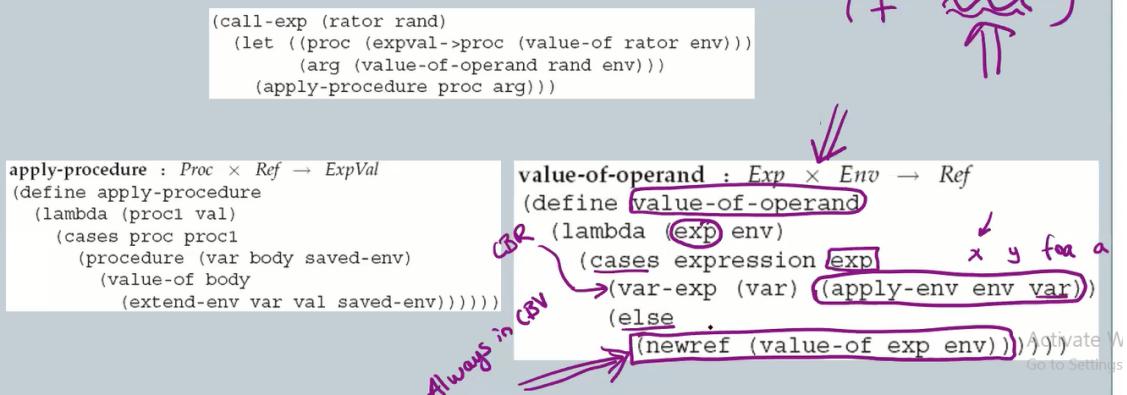
REF - OR

- 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 (value-of)



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



Above, when i CBR: if exp is a variable we know that it exists in the memory so we get the pointer that points to that expression from the environment

Delete the line labelled with CBR and tell us about CBV

- lazy evaluation says let's wait
- Metin Sezgin: In finding the value of the arguments until we really need to use it. Okay. Because it may be the case that we don't need it like this case over

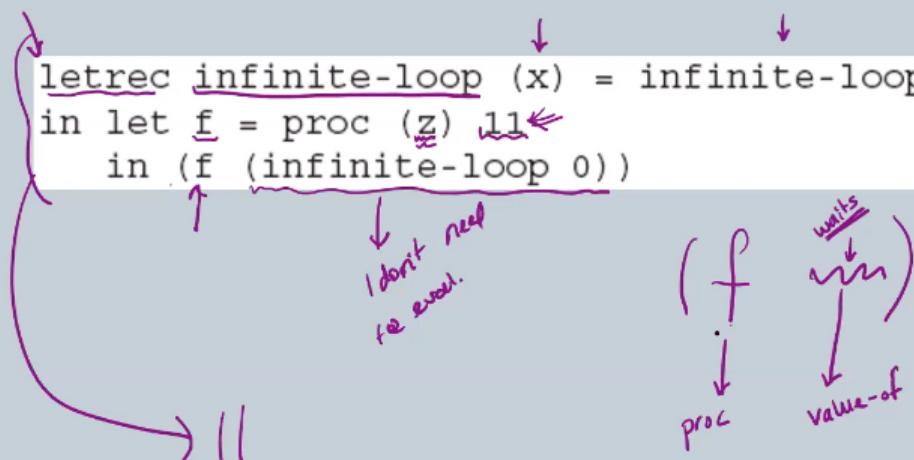
16:10

Lazy evaluation

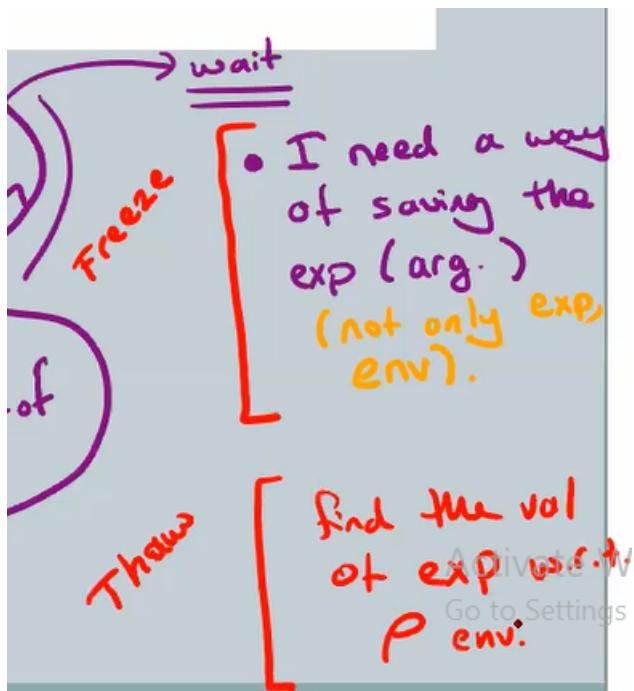


- Call-by-name ✓
- Call-by-need ✓

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



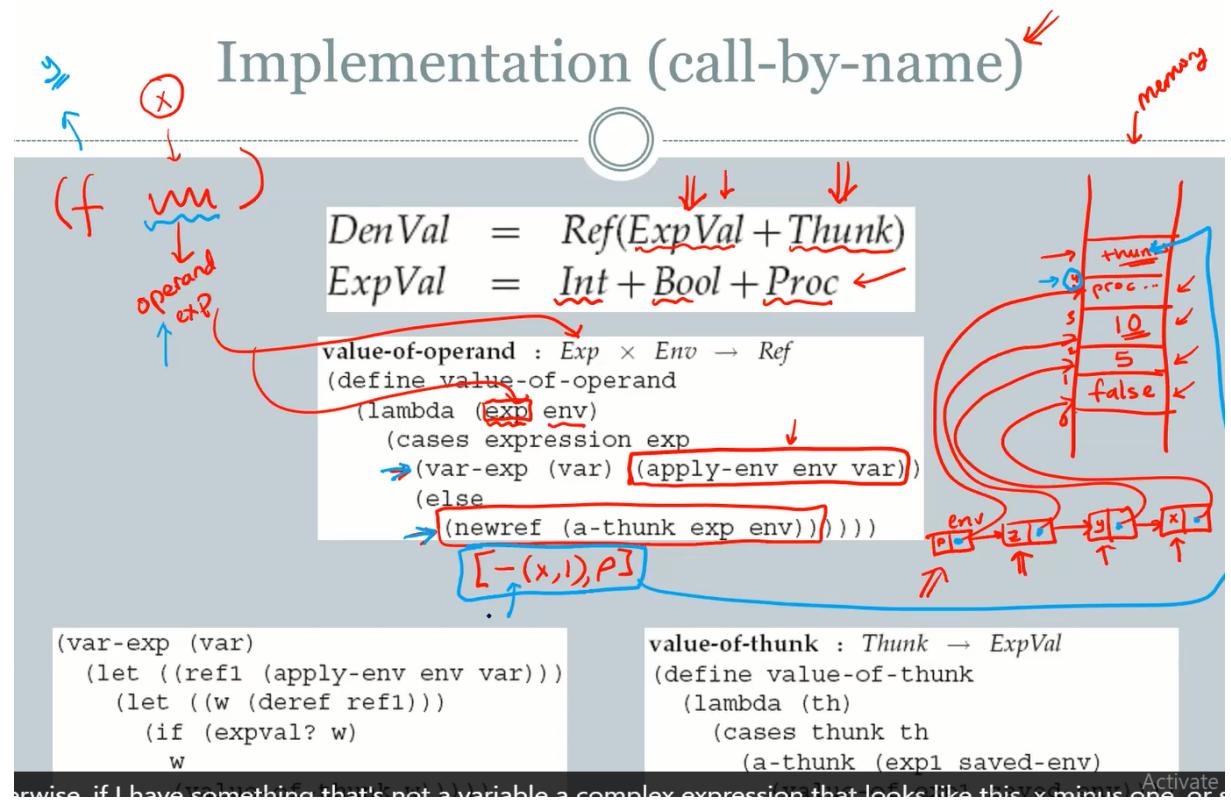
I need to save the argument which waits since I might need to evaluate and use it later on



```
(define-datatype thunk thunk?
  (a-thunk
    (exp1 expression?)
    (env environment?)))
```

IREF EXTENSION: call by name

Implementation (call-by-name) 



(+ 1 2)

DenVal = Ref(ExpVal + Thunk)
ExpVal = Int + Bool + Proc

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 (a-thunk exp env))))))`

value-of-thunk : Thunk → ExpVal
`(define value-of-thunk
 (lambda (th)
 (cases thunk th
 ((a-thunk (exp1 saved-env))`

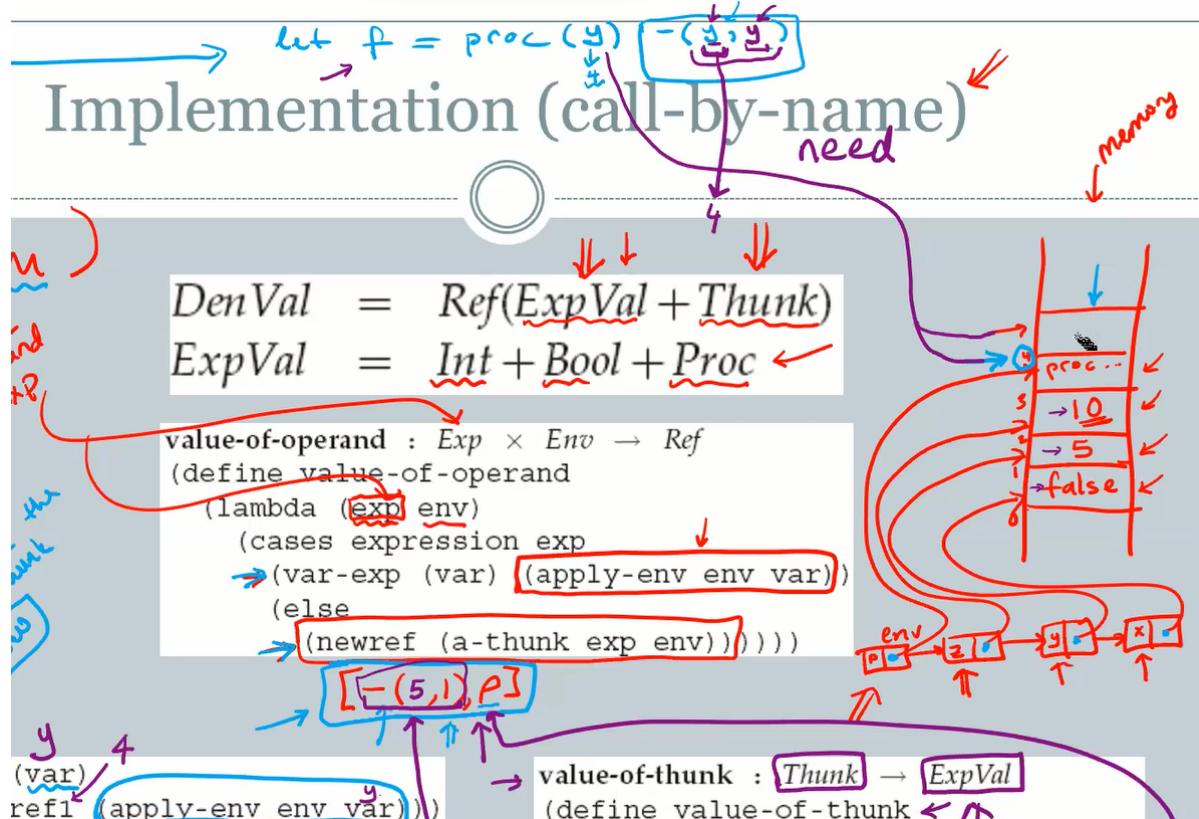
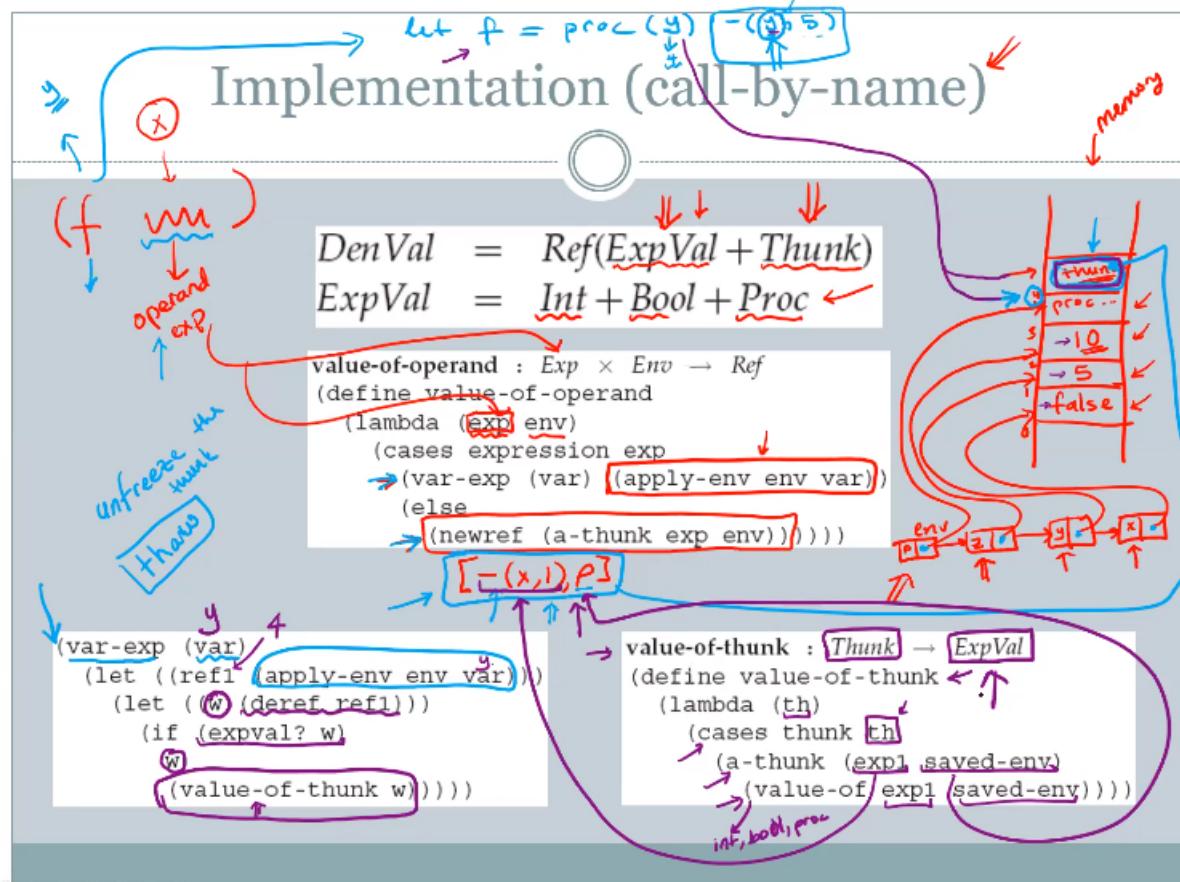
memory

Activate

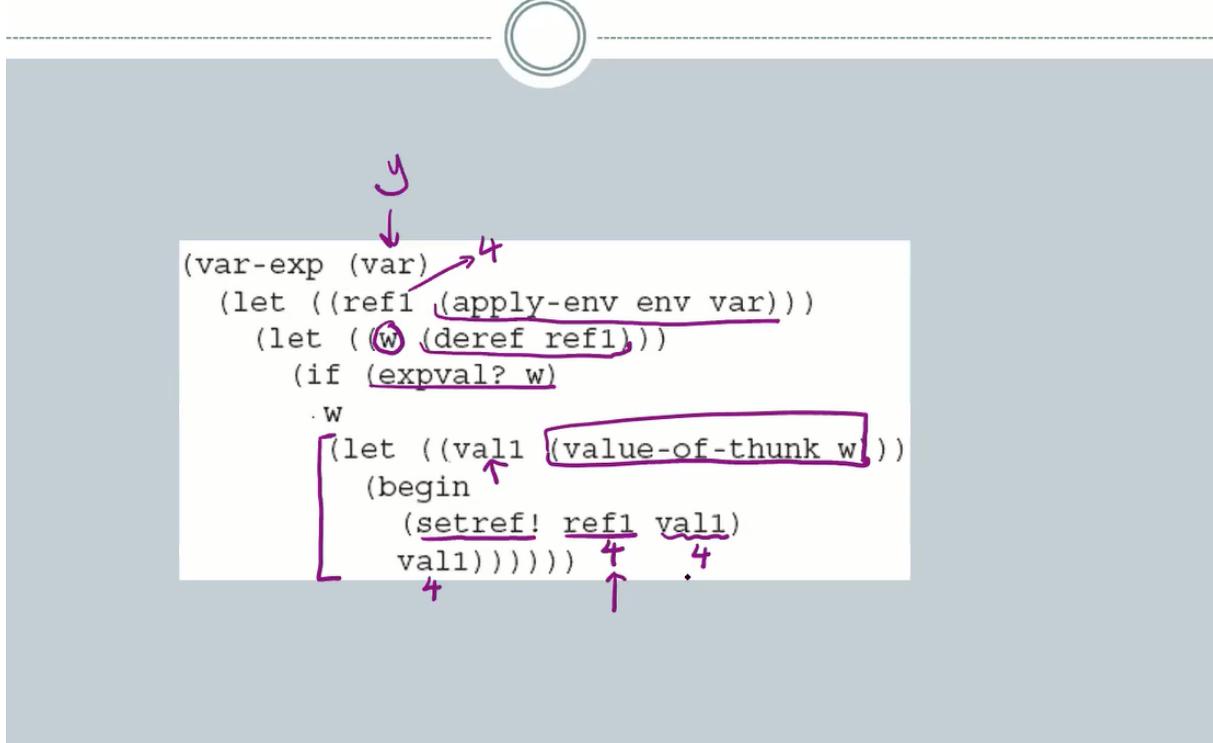
otherwise, if I have evaluated this, it's a variable, a complex expression, or that looks like this, then one or

Here for valueofoperand if we r looking for a vlaue of a plain variable like x then we simply return its address location this is implemented ad the varexp var line otherwise if we have anything else we freeze the operation adn the environement and put it in an address

Unfreeze when you see the variable y in this case this is called thawing



Memoization (call-by-need)



We have not only found value of the funk, but you've also set the contents of the memory sell for to that new value.

Recursive vs. Iterative Control Behavior

- Consider

```
(define fact
  (lambda (n)
    (if (zero? n) 1 (* n (fact (- n 1))))))
```

- The trace

```
(fact 4)
= (* 4 (fact 3))
= (* 4 (* 3 (fact 2)))
= (* 4 (* 3 (* 2 (fact 1))))
= (* 4 (* 3 (* 2 (* 1 (fact 0)))))
= (* 4 (* 3 (* 2 (* 1 1))))
= (* 4 (* 3 (* 2 1)))
= (* 4 (* 3 2))
= (* 4 6)
= 24
```

Activate
Go to Settings

Recursive vs. Iterative Control Behavior

- Consider

```
(define fact-iter
  (lambda (n)
    (fact-iter-acc n 1)))
(define fact-iter-acc
  (lambda (n a)
    (if (zero? n) a (fact-iter-acc (- n 1) (* n a)))))
```

- The trace

```
(fact-iter 4)
= (fact-iter-acc 4 1)
= (fact-iter-acc 3 4)
= (fact-iter-acc 2 12)
= (fact-iter-acc 1 24)
= (fact-iter-acc 0 24)
= 24
```

iterative.

Activ
Go to

- Consider

```
(define fact
  (lambda (n)
    (if (zero? n) 1
        * n (fact (- n 1))))))
```

this needs remebering

- The trace

ider

```
(define fact
  (lambda (n)
    (if (zero? n) 1
        * n (fact (- n 1))))))
```

trace

```
(fact 4)
= (* 4 (fact 3))
= (* 4 (* 3 (fact 2)))
= (* 4 (* 3 (* 2 (fact 1))))
```

fact-c

$\hat{=} \boxed{\text{proc}}$

$$\begin{cases} \text{proc} \\ \downarrow \\ (\lambda x) \\ (* x n) \\ (n-1)! \cdot n \\ = n! \end{cases}$$

- the continuation of this procedure will know how to take

16:03

- Metin Sezgin: The answer, coming from a recursive call and convert it into the final answer and here's my definition. I will say is

Continuation is a procedure which will take the result of factorial computed on $x-1$ and it should know how to convert that to factorial of x .

```

; (factr 10)

(define (fact-c x cont)
  (if (= x 0) (cont 1)
      (fact-c (- x 1) (lambda (n) (cont (* x n))))))

(trace fact-c)

(fact-c 5 (lambda (x) x))

```

Welcome to DrRacket, version 7.1 [3m].
 Language: racket, with debugging [custom]; memory limit: 128 MB.

```

> (fact-c 5 #<procedure>
> (fact-c 4 #<procedure>
> (fact-c 3 #<procedure>
> (fact-c 2 #<procedure>
> (fact-c 1 #<procedure>
> (fact-c 0 #<procedure>
<120
120
>

```

Alternative implementation

- Consider

```

(define (fact x) (fact-c x (lambda (x) x)))

```

```

(define (fact-c x cont)
  (if (= x 0)
      (cont 1)
      (fact-c (- x 1) (lambda (n) (cont (* x n))))))

```

- The trace

```

> (fact 4)
> (fact-c 4 #<procedure:...33/fact-cont.rkt:6:27>)
> (fact-c 3 #<procedure:...33/fact-cont.rkt:11:22>)
> (fact-c 2 #<procedure:...33/fact-cont.rkt:11:22>)

```

AAAAAAAAAA

WRITE A TAIL RECURSIVE VERSION OF A OVER B EXERCISEEEE

A CPS Interpreter

- The environment grows as we evaluate expressions
- Now we need to keep around a list of things to do after the evaluation of each expression.
- Introduce apply-cont ↗
 - Example:

```
FinalAnswer = ExpVal
apply-cont : Cont × ExpVal → FinalAnswer
  (apply-cont (end-cont) val)
  = (begin
      ↑ end-cont
    → (eopl:printf "End of computation.\n")
      val)
```

AAAAA

Below take the expression and apply the continuation to it ie when 7 apply it to a num value

Evaluation

• Value-of-program

```
value-of-program : Program → FinalAnswer
(define value-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1))
      (value-of/k exp1 (init-env) (end-cont))))))
```

value-of with cont.

• Value-of/k

```
value-of/k : Exp × Env × Cont → FinalAnswer
(define value-of/k
  (lambda (exp env cont)
    (cases expression exp
      (const-exp (num) (apply-cont cont (num-val num)))
      (var-exp (var) (apply-cont cont (apply-env env var)))
      (proc-exp (var body)
        (apply-cont cont
          (proc-val (procedure var body env))))
      ...)))
```

7
value stored in the env.

FOR THE ZERO?:

We take zero? <exp>

Evaluate the expression to get its value and then feed it to the zero1-cont exp1 = val

The boolean expressed value gets passed to the continuation

Evaluation

- Letrec

```
(letrec-exp (p-name b-var p-body letrec-body)
  (value-of/k letrec-body)
  (extend-env-rec p-name b-var p-body env)
  (cont))
```

- Zero?

```
(zero?-exp (exp1)
  (value-of/k exp1 env)
  (zerol-cont cont))
= (apply-cont (zerol-cont cont) val)
  (bool-val
    (zero? (expval->num val))))
```

letrec foo(x) = ---
in body
if

zero?(<exp1>) → true if
false if

Activate
Go to Settings

- I can kind of see what needs to be remembered as I evaluate a lot expression as I evaluate a lot expression. First I have to evaluate

39:09

- Metin Sezgin: This expression and find that its value is 10 and then I have to remember that I have to set x equals two to 10 and then also have to remember that I have to find the value of the body.

39:22

- Metin Sezgin: In that environment, right. So there are multiple things that I need to remember:

Let exp cont:

- The rest of the computation for a let is to find the value of the body so

41:03

- Metin Sezgin: Really this lead expression continuation will just find the value of the body in an environment that's been extended with x equals whatever value comes out of

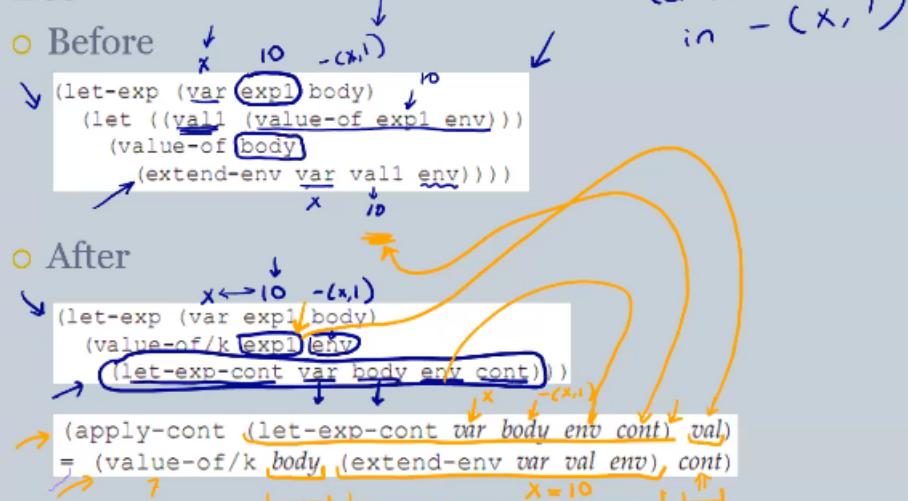
41:12

- BURCU YILDIZ: Here, right.

At the end it sends it to cont (the next continuation):

Evaluation

- Let



For if the main idea is it looks at the condition and THEN applies the continuation which is either the computation of exp2 or exp3 depending on the exp1 and whatever result comes out is passed to the cont being passed inside the below function which is the first continuation

The result of the test true or false is passed as a continuation here:

Example

()

```

(value-of/k <<letrec p(x) = x in if b then 3 else 4>>
  ρ0 cont0)
= letting ρ1 be (extend-env-rec ... ρ0)
(value-of/k <<if b then 3 else 4>> ρ1 cont0)
= next, evaluate the test expression
(value-of/k <<b>> ρ1 (test-cont <<3>> <<4>> ρ1 cont0))
= send the value of b to the continuation
(apply-cont (test-cont <<3>> <<4>> ρ1 cont0)
  (bool-val #t))
= evaluate the then-expression
(value-of/k <<3>> ρ1 cont0)
= send the value of the expression to the continuation
(apply-cont cont0 (num-val 3))
= invoke the final continuation with the final answer
  (begin (eopl:printf ...) (num-val 3))

```

3

- Zero?

```

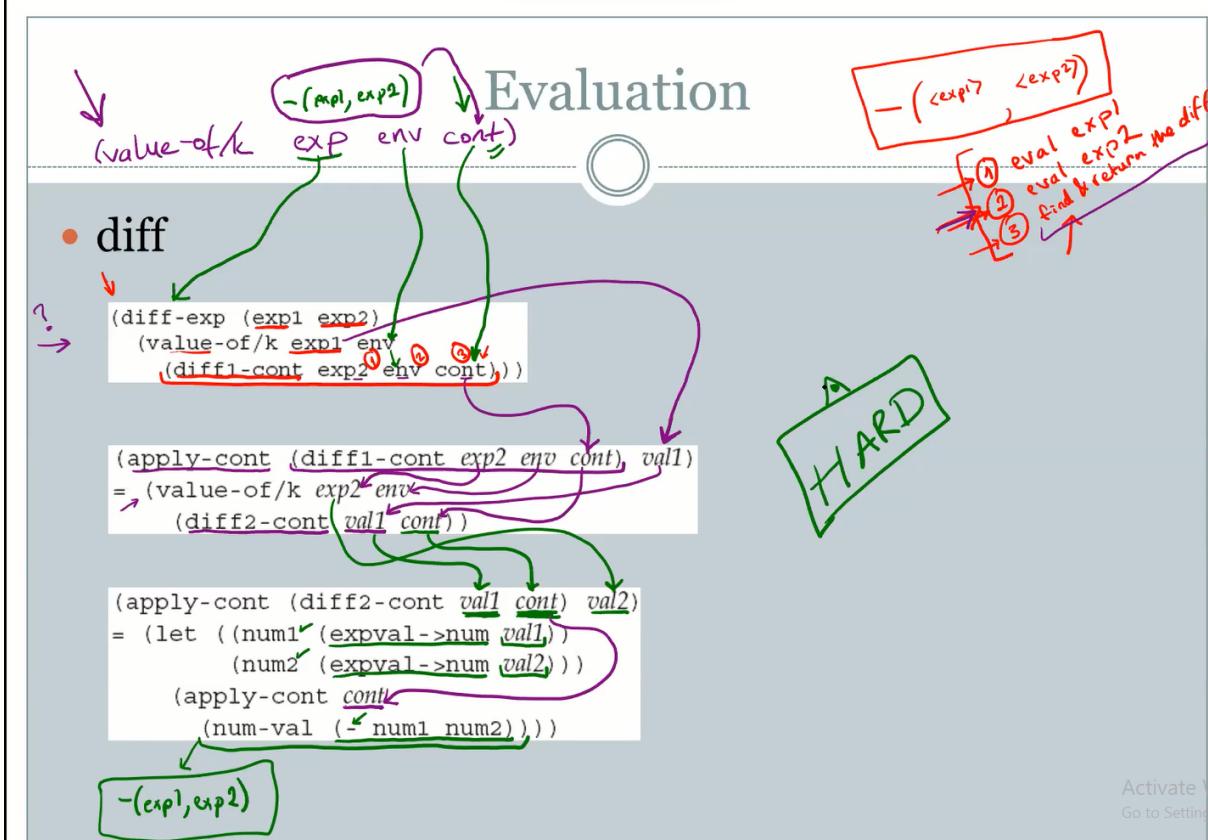
(zero?-exp (exp1)
  (value-of/k exp1 env
    (zerol-cont cont)))

```

zero?(x) ↗ 5 ↘ #f

→ (apply-cont (zerol-cont cont) val) ↗
 = (apply-cont cont)
 (bool-val
 (zero? (expval->num val))) ↘

THE STEPS ARE ALWAYS ABOUT WHAT TO DO DIO NEXT SO FOR DIFF WE FIND EXP1 THEN EXP2 THEN EVALUATE THE SUBTRACTION



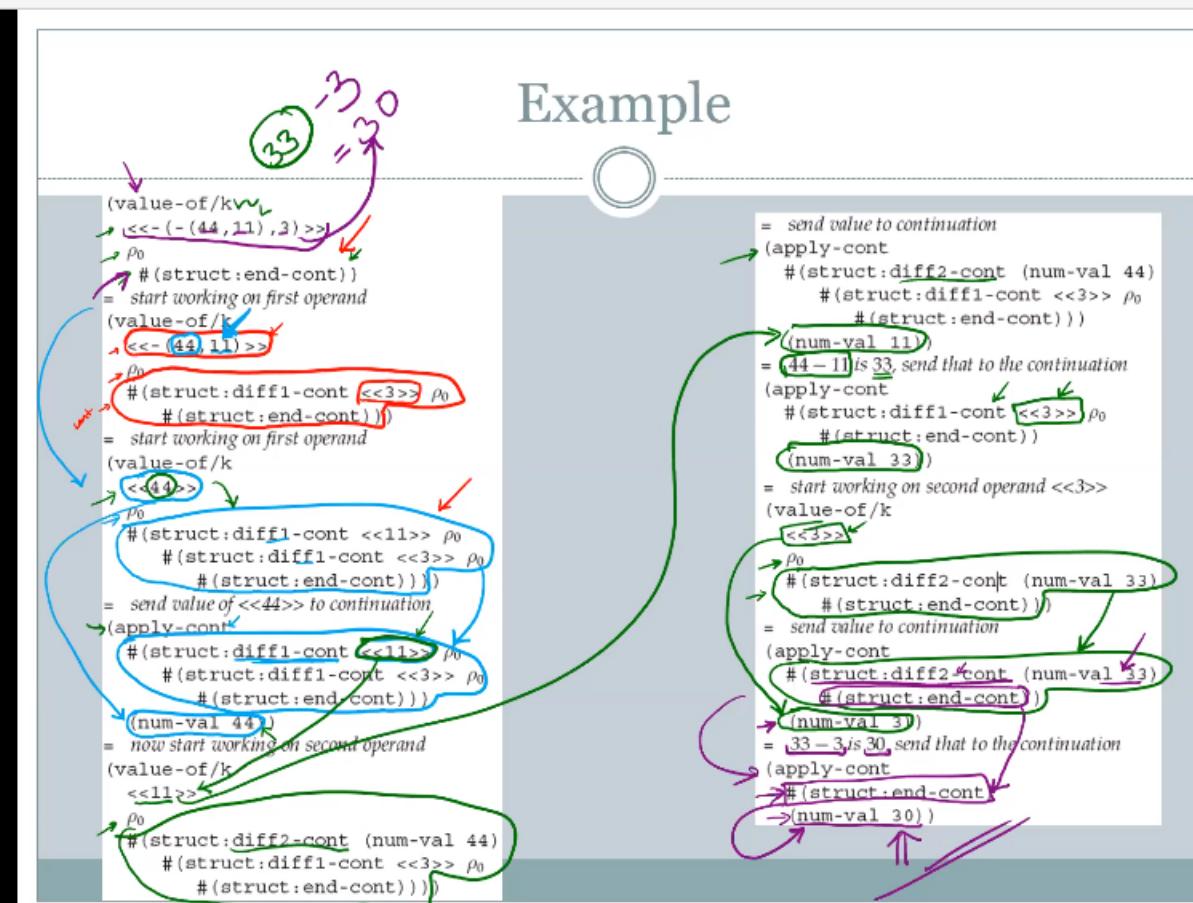
AAA

:

HERE BELOW, THE BELOW SECTION WITH THE STRUCTS REPRESENTS WHAT TO DO NEXT IN ORDER SO IN FIRST BLUE CASE OF IT YOU EVALUATE44 THEN THE NEXT THING TO DO IS TO EVALUATE 11 AND

THEN 3

Example



Diff1continuation = evaluate the second expression

Diff2continuation = actually perform the subtraction operation

Evaluation

• Procedure application

○ Before

```
(call-exp (rator rand)
  (let ((proc1 (expval->proc (value-of rator env)))
    (val (value-of rand env)))
  (apply-procedure (proc1 val))))
```

○ After

```
(call-exp (rator rand)
  (value-of/k rator env
    (rator-cont rand env cont)))
= (apply-cont (rator-cont rand env cont) val1)
= (value-of/k rand env
  (rand-cont val1 cont))
= (apply-cont (rand-cont val1 cont) val2)
= (let ((proc1 (expval->proc val1)))
  (apply-procedure/k proc1 val2 cont))
```

apply-procedure/k : Proc × ExpVal × Cont → FinalAnswer

```
(define apply-procedure/k
  (lambda (proc val cont)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of/k body)
        (extend-env var val saved-env))))
```

Activate W
Go to Settings