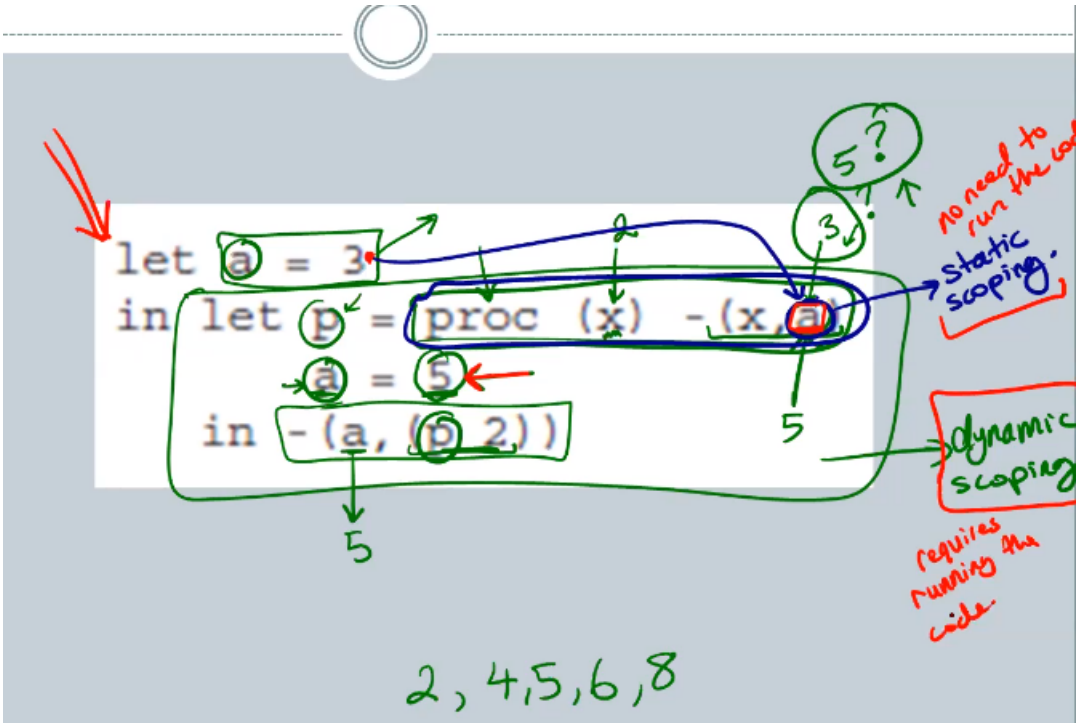
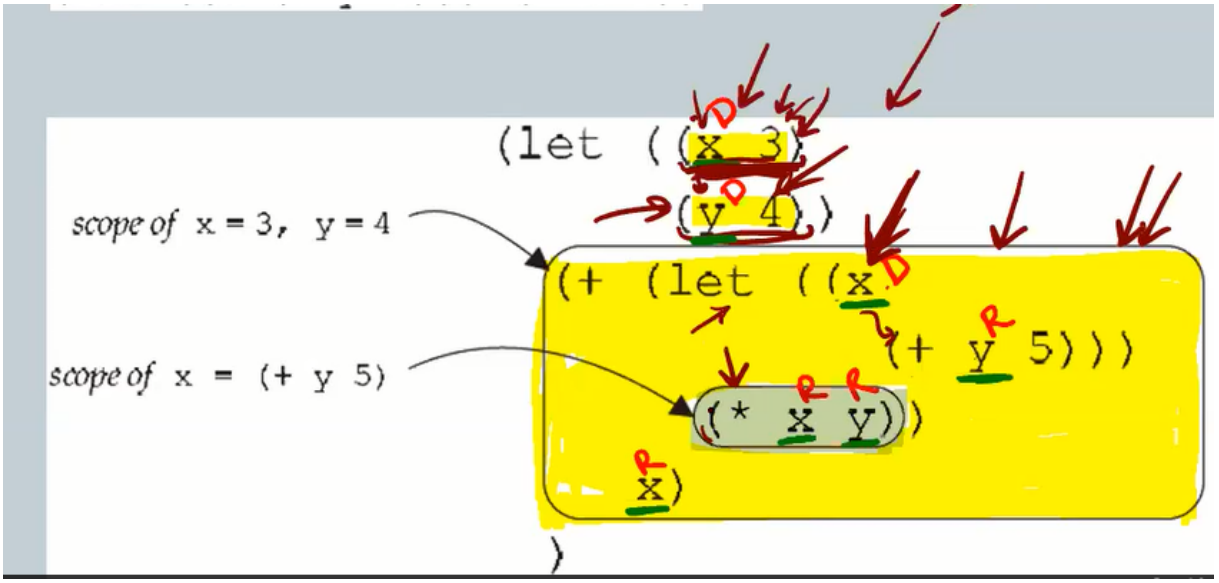
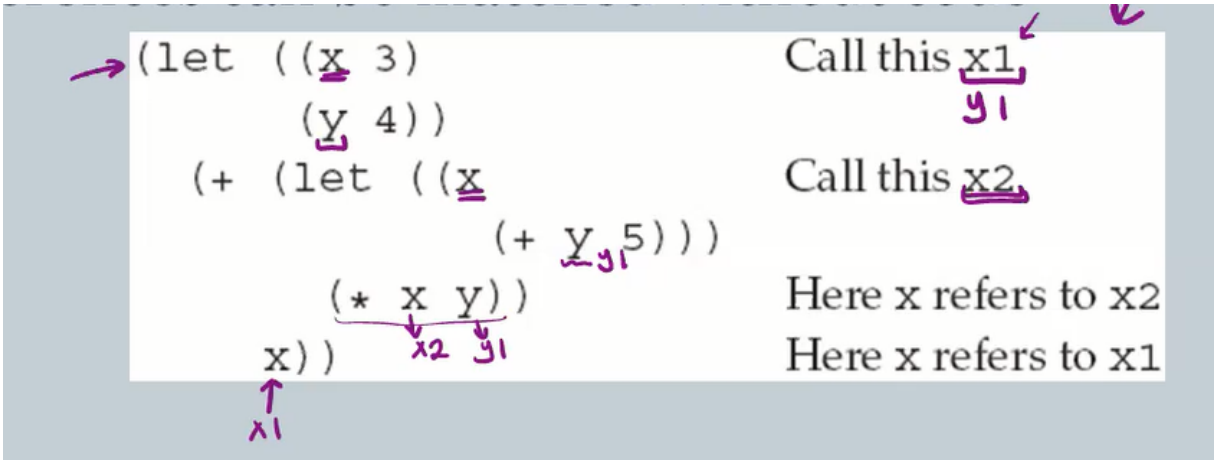
Lecture 15

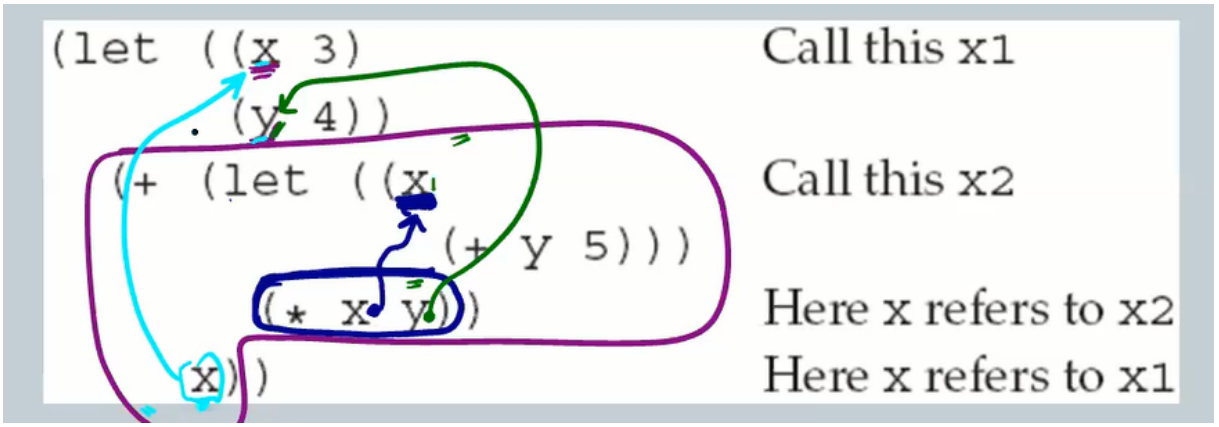


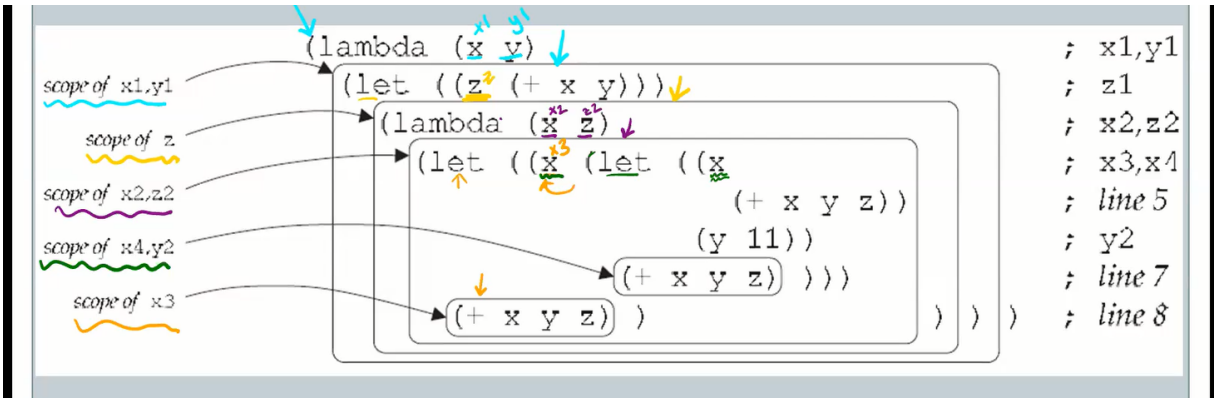
Static scoping takes the previous declared value above its level

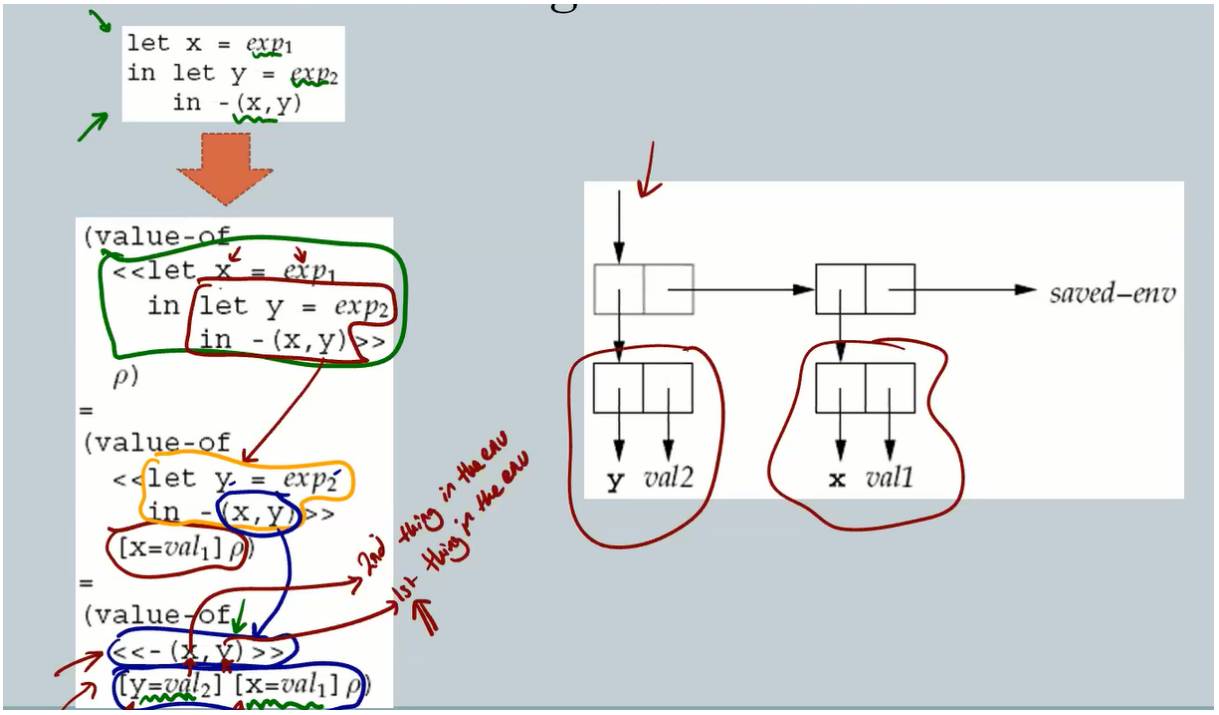
Dynamic scoping takes the last declared value of a

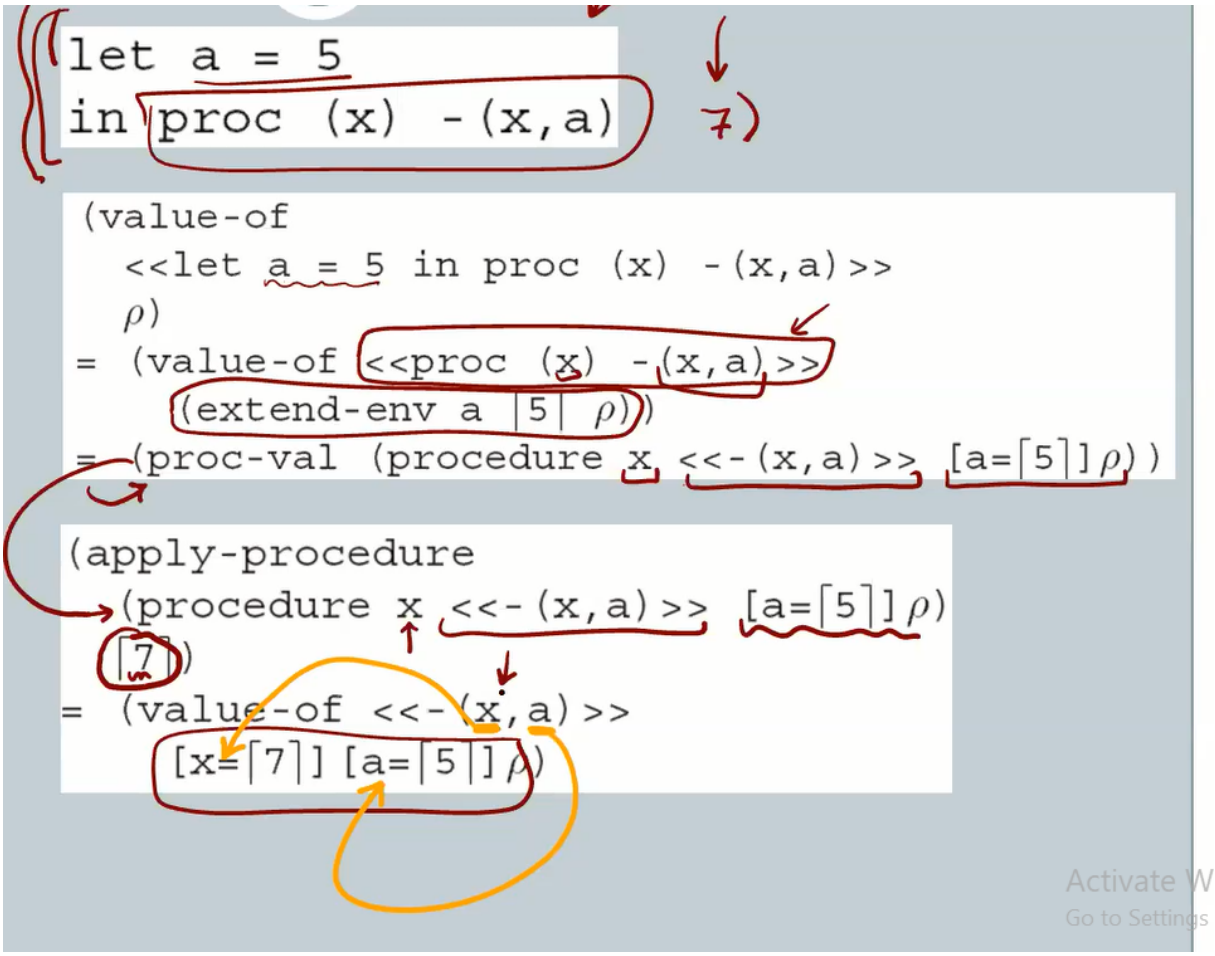



(let ((x 3)
(y 4))
(+ (let ((x
(+ y 5)))
(* x y))
x))
Call this x1
Call this x2
Here x refers to x2
Here x refers to x1


(let ((x 3)
(y 4))
(+ (let ((x
(+ y 5)))
(* x y))
x))
Call this x1
Call this x2
Here x refers to x2
Here x refers to x1

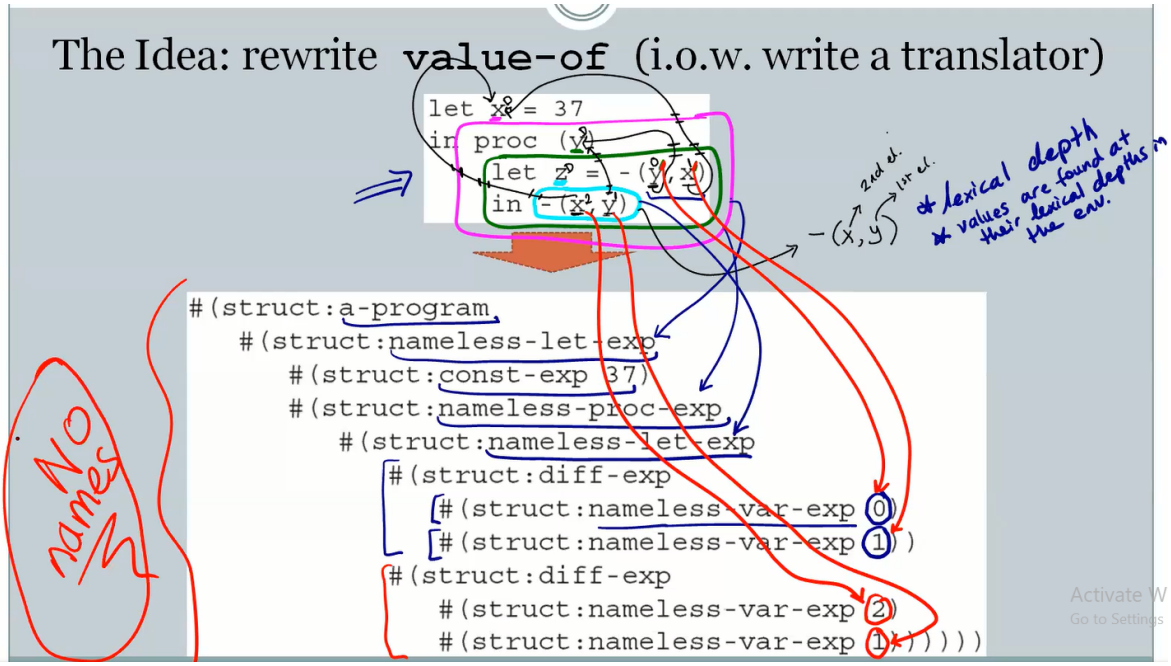

(lambda (x y)
(let ((z (+ x y)))
(lambda (x z)
(let ((x (let ((x
(+ x y z))
(y 11))
(+ x y z)))
(+ x y z))
)))
; x1,y1
; z1
; x2,z2
; x3,x4
; line 5
; y2
; line 7
; line 8
scope of x1,y1
scope of z
scope of x2,z2
scope of x4,y2
scope of x3





LEXICAL DEPTH:

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



The translator: the target language

lexical address $X \to 0$

Expression ::= `%lexref` *number*

`nameless-var-exp (num)`

Expression ::= `%let` *Expression* `in` *Expression*

`nameless-let-exp (exp1 body)`

Expression ::= `%lexproc` *Expression*

`nameless-proc-exp (body)`

var-exp

let x = ...

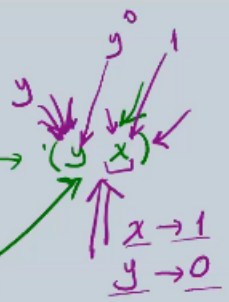
in ...

proc (x) body

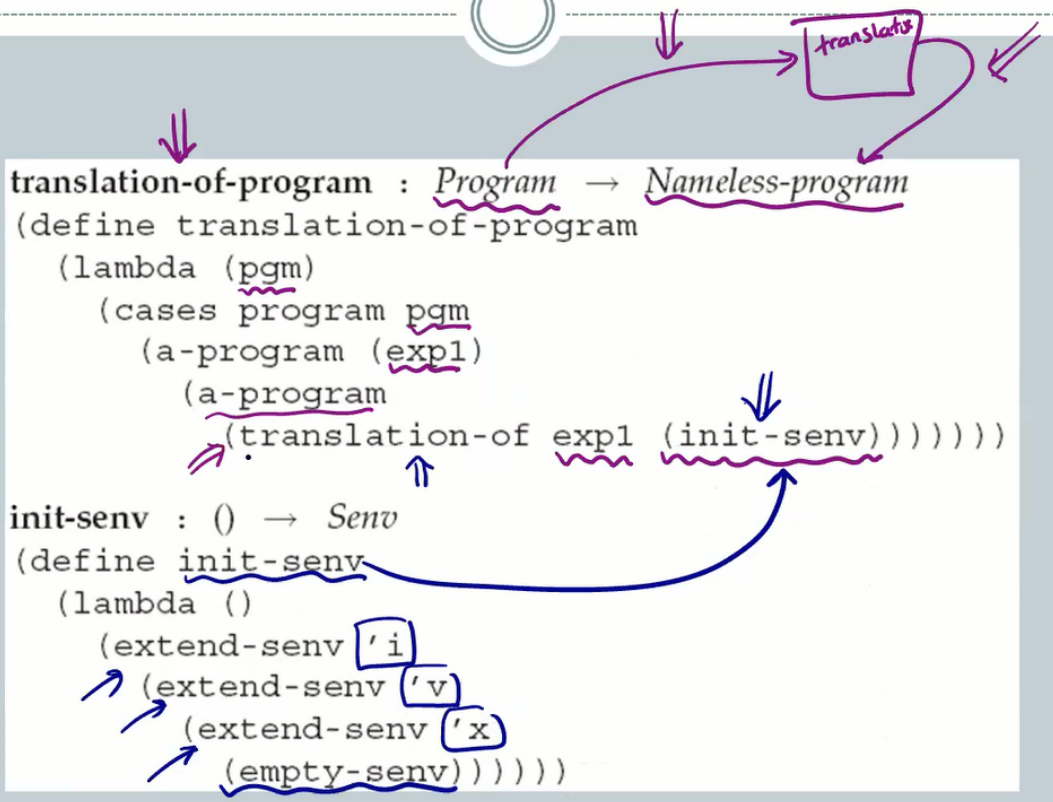
The translator: Exp x Senv → NamelessExp

Static Environment

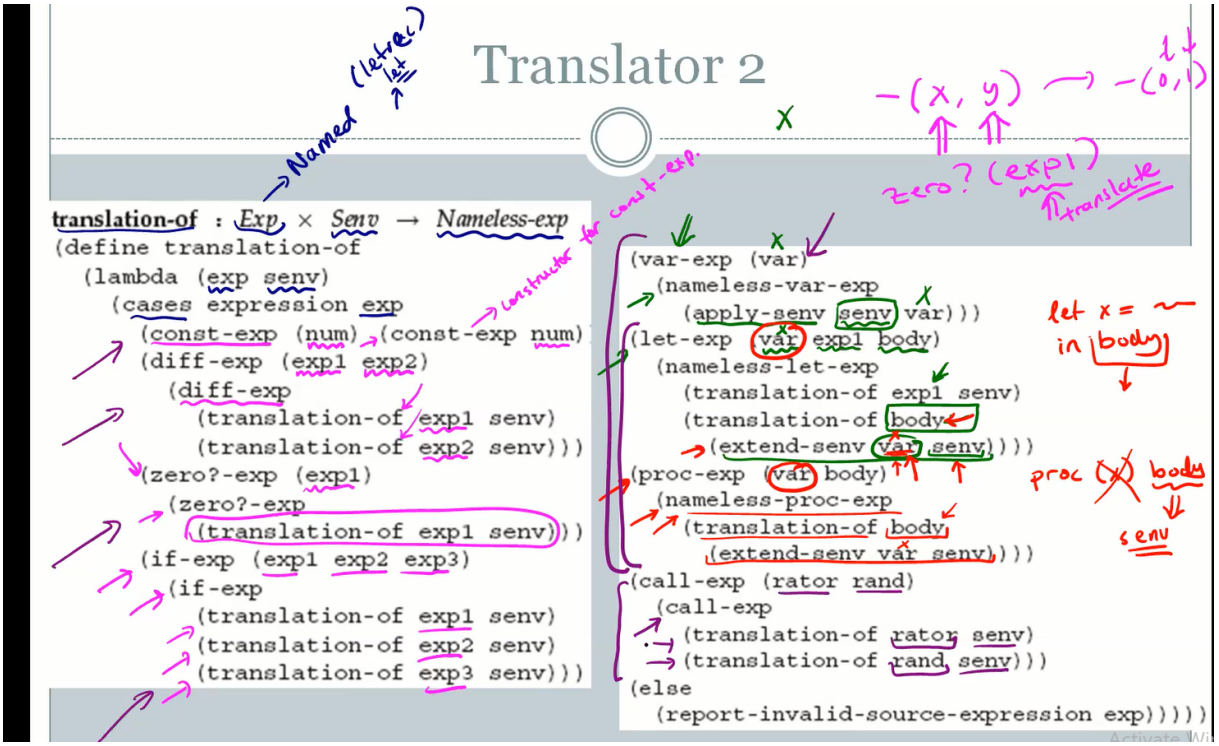
```
Senv = Listof(Sym)
Lexaddr = N

empty-senv : () → Senv
(define empty-senv
  (lambda ()
    '()))

extend-senv : Var × Senv → Senv
(define extend-senv
  (lambda (var senv)
    (cons var senv)))

apply-senv : Senv × Var → Lexaddr
(define apply-senv
  (lambda (senv var)
    (cond
      ((null? senv)
       (report-unbound-var var))
      ((eqv? var (car senv))
       0)
      (else
       (+ 1 (apply-senv (cdr senv) var))))))
```



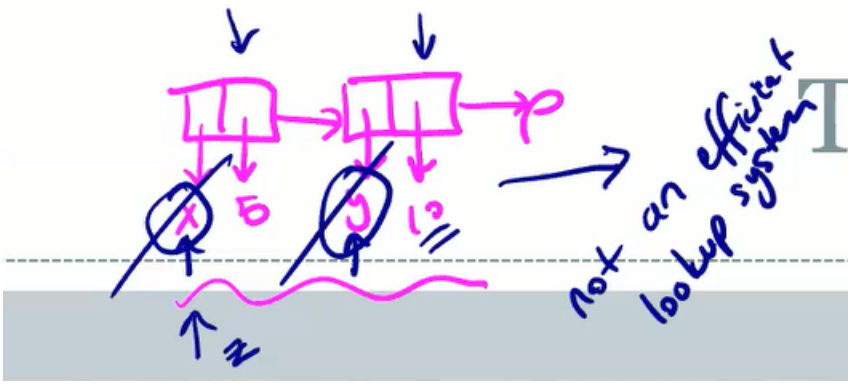
Translator 1



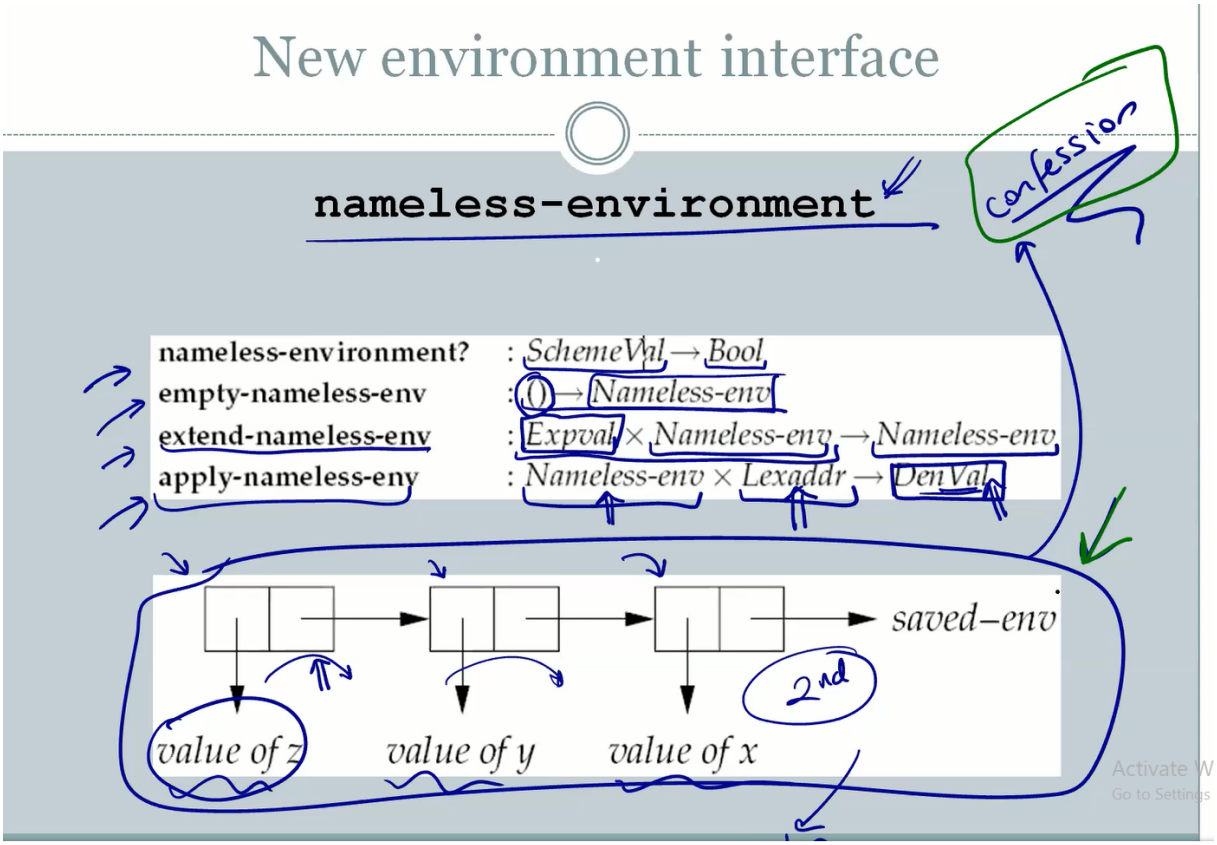

```
translation-of-program : Program → Nameless-program
(define translation-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (a-program
          (translation-of exp1 (init-senv)))))))

init-senv : () → Senv
(define init-senv
  (lambda ()
    (extend-senv 'i
      (extend-senv 'v
        (extend-senv 'x
          (empty-senv))))))
```

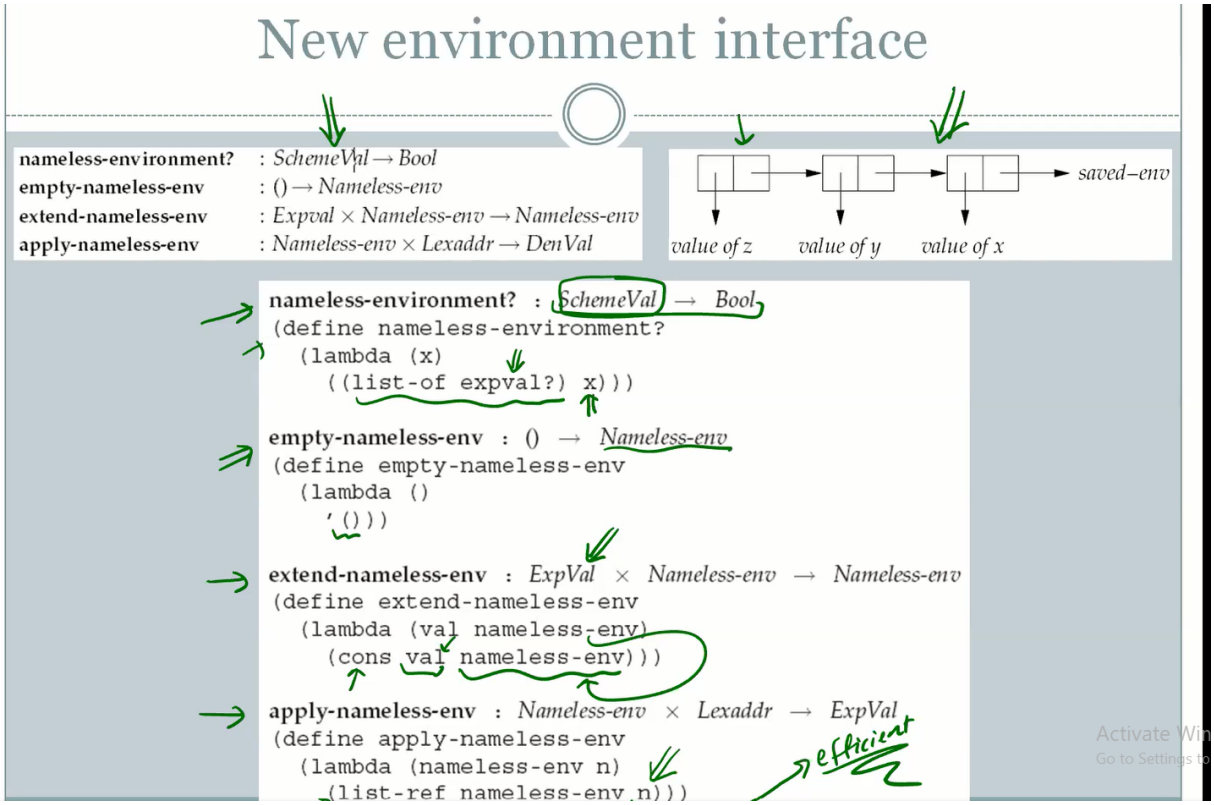


BY MAKING TH EPROGRAM NAMELESS WE MAKE THE INTERPRETATION OR EXECUTION OF THE PROGRAM SIMPLER SINCE INSTEAD OF LOOKING FOR Z ONE BY ONE THROUGH THE LEVELS INSIDE A PROGRAM BELOW LIKE A LINKED LIST WE DIRECTLY GO TO THE ADDRESS

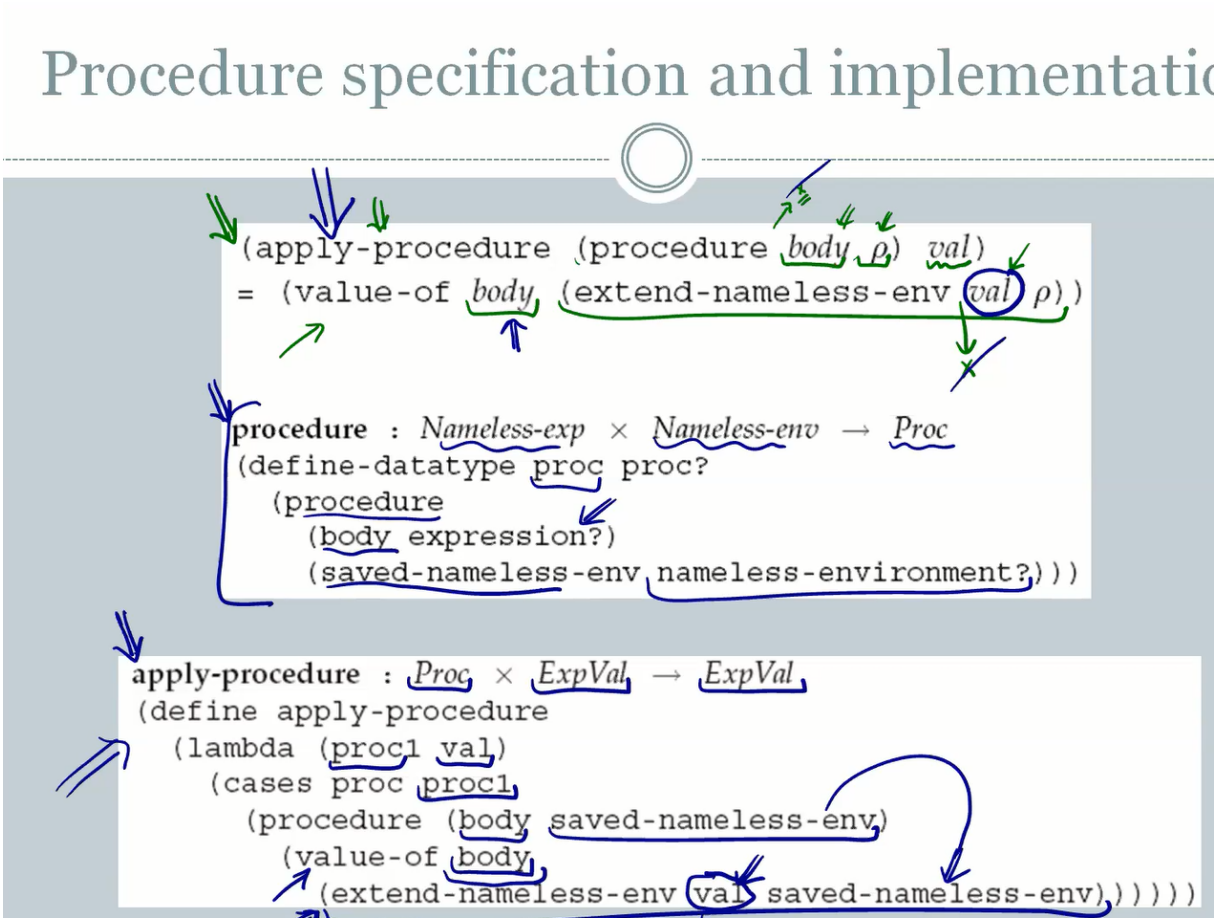


THE TRANSLATOR TRANSLATES IT TO A NAMELESS VERSION AND FEEDS IT TO VALUE OF WHICH IS THE INTERPRETER SO NOW WE NEED TO HAVE A NEW INTERPRETER WHERE IT TAKES IN NAMELESS


New environment interface
nameless-environment
Confession
nameless-environment? : SchemeVal → Bool
empty-nameless-env : () → Nameless-env
extend-nameless-env : Expval × Nameless-env → Nameless-env
apply-nameless-env : Nameless-env × Lexaddr → DenVal
saved–env
2nd
value of z
value of y
value of x
Activate W
Go to Settings


New environment interface
nameless-environment? : SchemeVal → Bool
empty-nameless-env : () → Nameless-env
extend-nameless-env : Expval × Nameless-env → Nameless-env
apply-nameless-env : Nameless-env × Lexaddr → DenVal
saved–env
value of z
value of y
value of x
nameless-environment? : SchemeVal → Bool
(define nameless-environment?
(lambda (x)
((list-of expval?) x)))
empty-nameless-env : () → Nameless-env
(define empty-nameless-env
(lambda ()
'()))
extend-nameless-env : ExpVal × Nameless-env → Nameless-env
(define extend-nameless-env
(lambda (val nameless-env)
(cons val nameless-env)))
apply-nameless-env : Nameless-env × Lexaddr → ExpVal
(define apply-nameless-env
(lambda (nameless-env n)
(list-ref nameless-env n)))
efficient
Activate Win
Go to Settings to

WHAT IS THIS:



LOOK AGAIN AT HOW PROC IS APPLIED

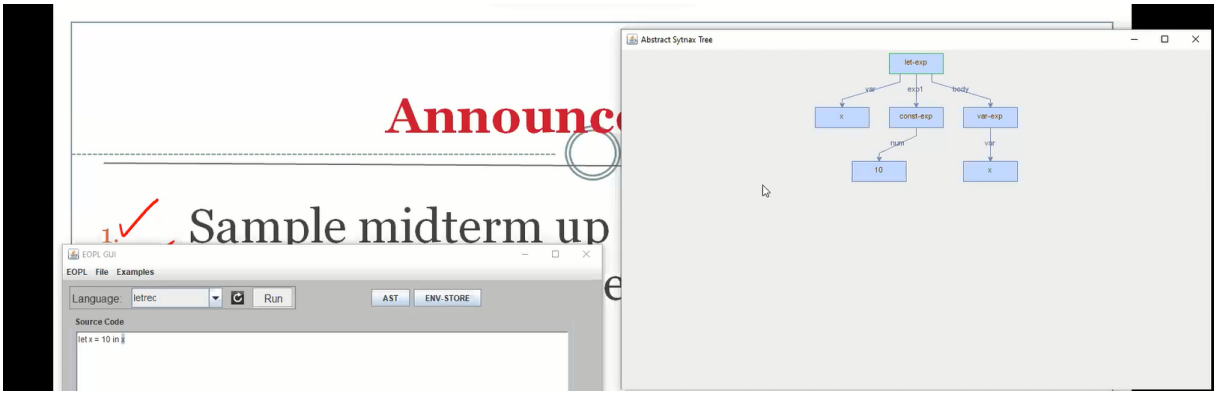
VALUEOF:

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.

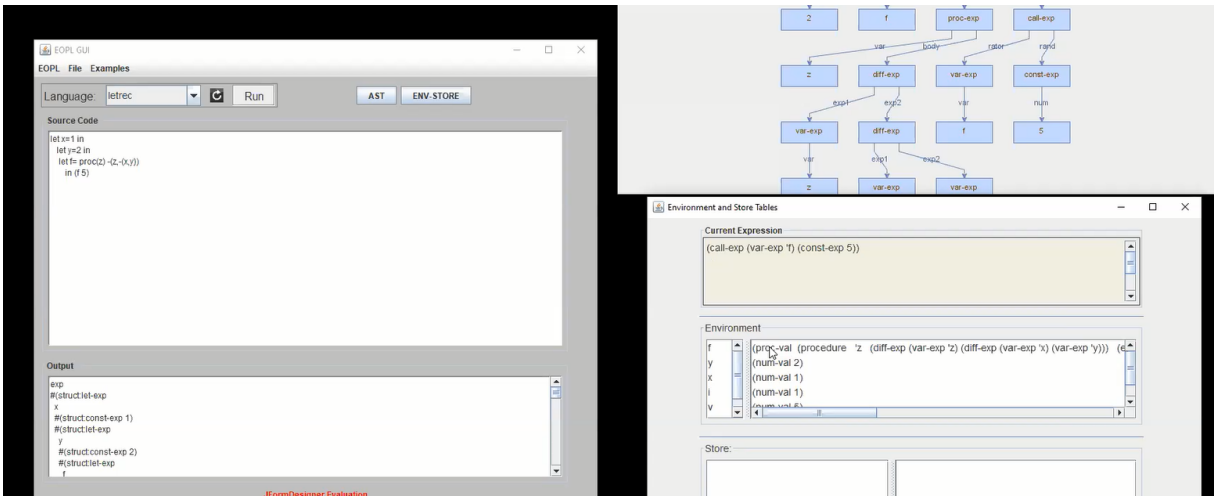
FOR LET THE IDEA IS TO FEED THE BODY TO THE VALUE OF WHERE WE ADD ANOTHER PARAMATER LIKE X = 12 CHECK THE SLIDE WITH ;ECTURE 12- REVIEW LET THIS TIME WE SIMPLY ADD 12 AND THAT IS IT:

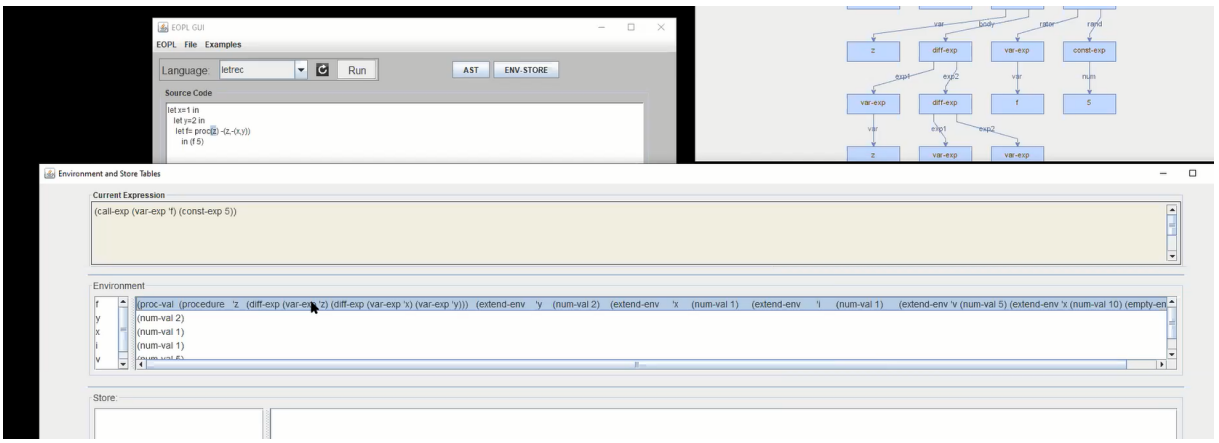
Interpreter for the new language

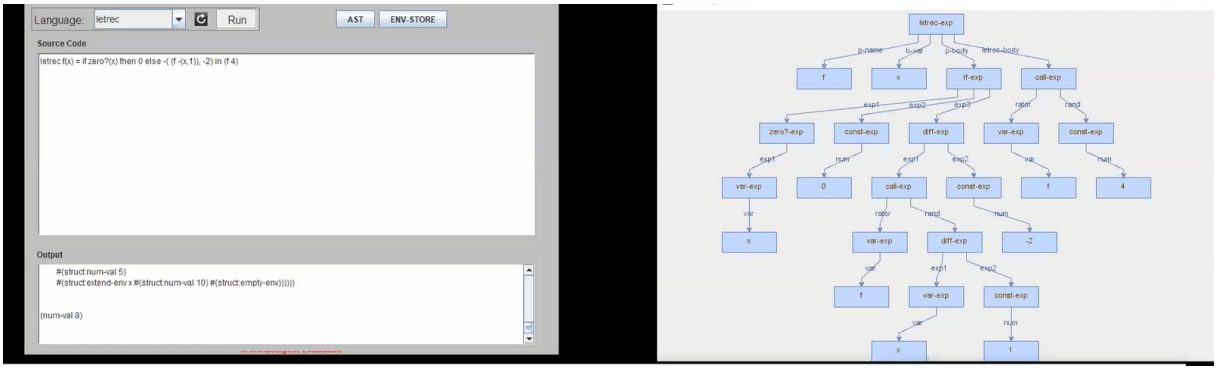
```
value-of : Nameless-exp × Nameless-env → ExpVal
(define value-of
  (lambda (exp nameless-env)
    (cases expression exp

      (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))

      (nameless-let-exp (exp1 body)
        (let ((val (value-of exp1 nameless-env)))
          (value-of body
            (extend-nameless-env val nameless-env))))

      (nameless-proc-exp (body)
        (proc-val
          (procedure body nameless-env)))

      (else
        (report-invalid-translated-expression exp)))))
```

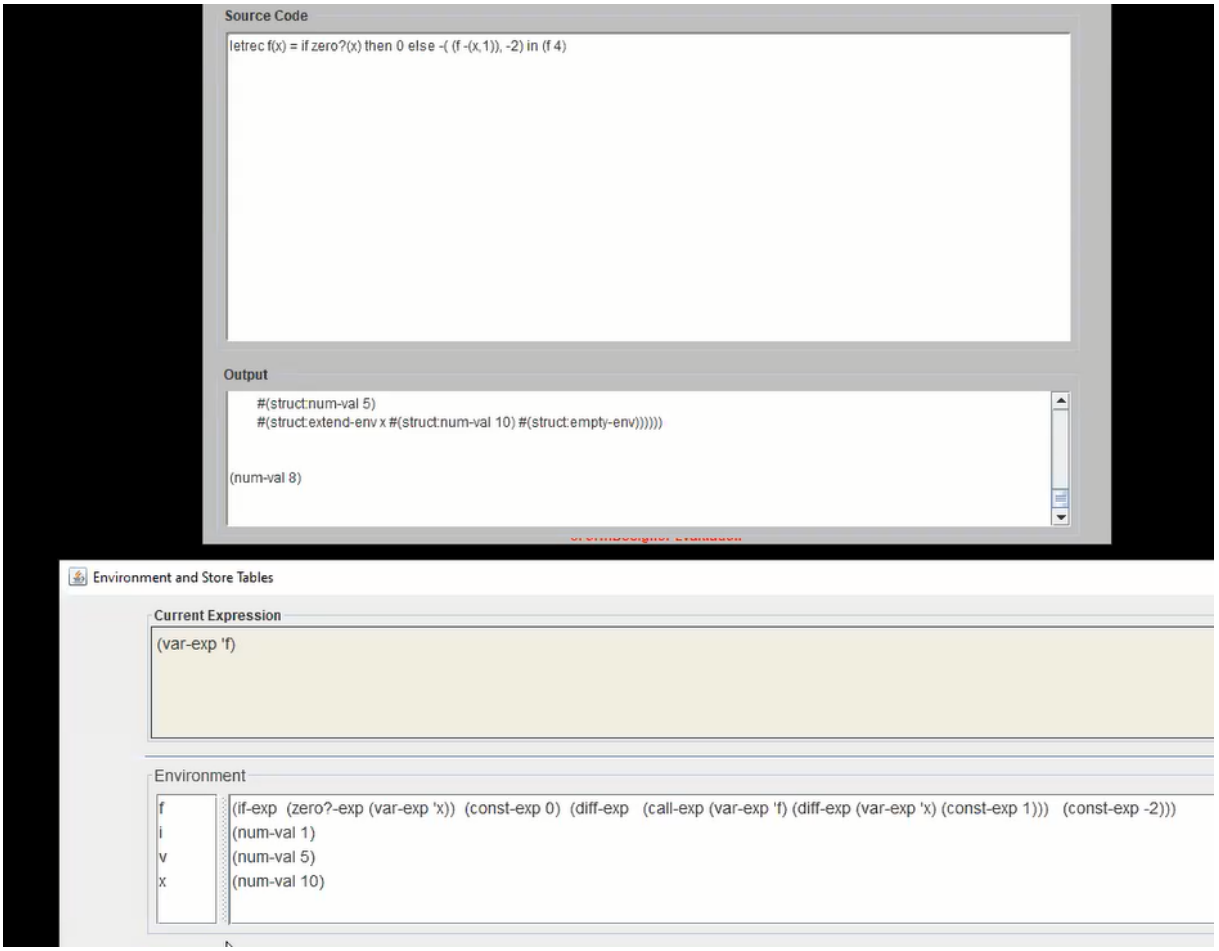



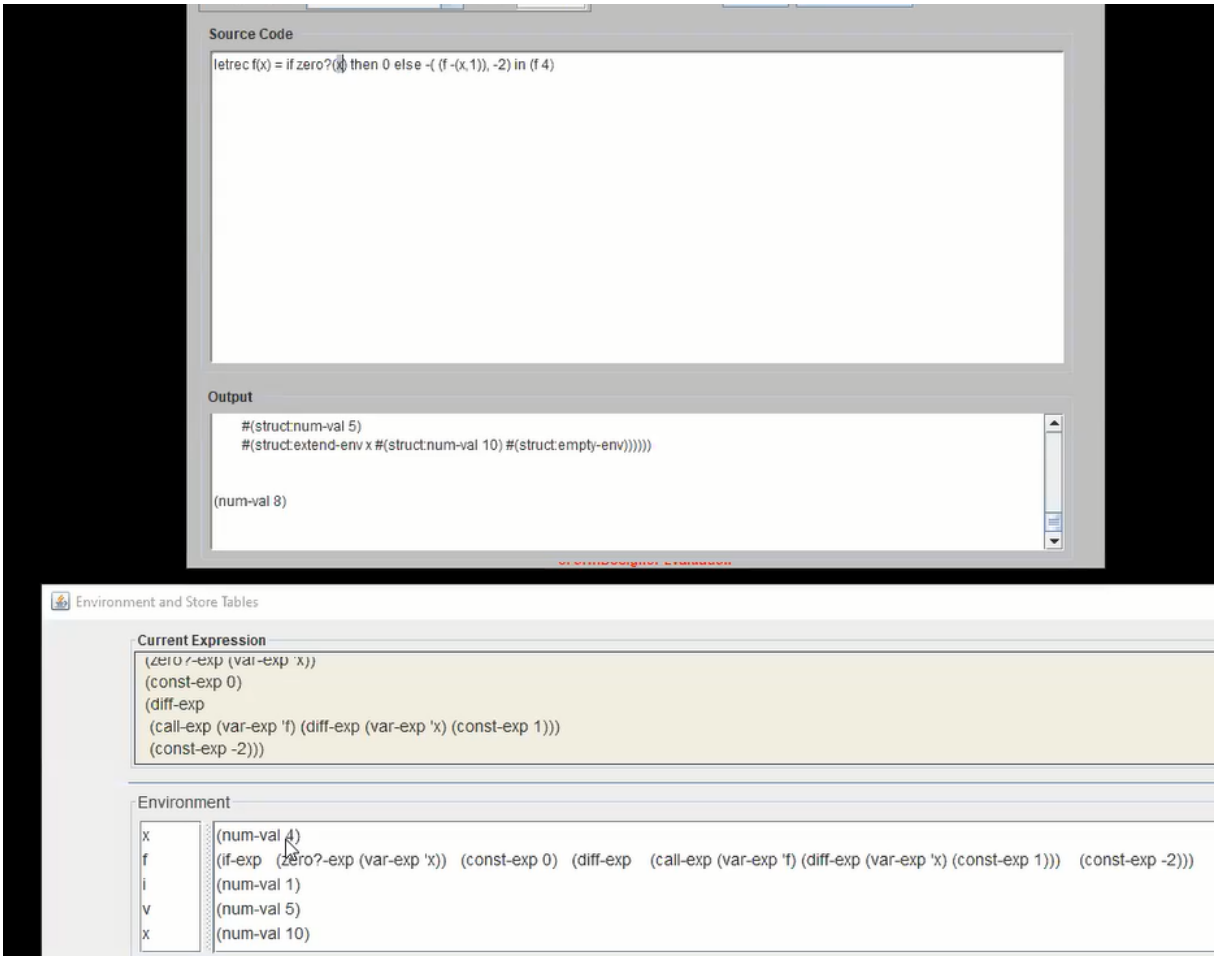

Announc
Sample midterm up
EOPL GUI
EOPL File Examples
Language: letrec
Run
AST
ENV-STORE
Source Code
let x=5 in
let y=3 in
let f=proc (x) -(x,1)in
(f y)
let-exp
var
exp1
body
x
const-exp
let-exp
num
y
const-exp
let-exp
3
f
proc-exp
call-exp
rator
rand
diff-exp
var-exp
var-exp
exp2
const-exp
f
y
num
1


EOPL GUI
EOPL File Examples
Language: letrec
Run
AST
ENV-STORE
Source Code
let x=1 in
let y=2 in
let f= proc(z) -(z,-(x,y))
in (f 5)
Output
exp
#(struct:let-exp
x
#(struct:const-exp 1)
#(struct:let-exp
y
#(struct:const-exp 2)
#(struct:let-exp
f
Environment and Store Tables
Current Expression
(call-exp (var-exp 'f) (const-exp 5))
Environment
(num-val 2)
(num-val 1)
(num-val 1)
Store:


EOPL GUI
EOPL File Examples
Language: letrec
Run
AST
ENV-STORE
Source Code
let x=1 in
let y=2 in
let f= proc(z) -(z,-(x,y))
in (f 5)
Environment and Store Tables
Current Expression
(call-exp (var-exp 'f) (const-exp 5))
Environment
(num-val 2)
(num-val 1)
(num-val 1)
Store:

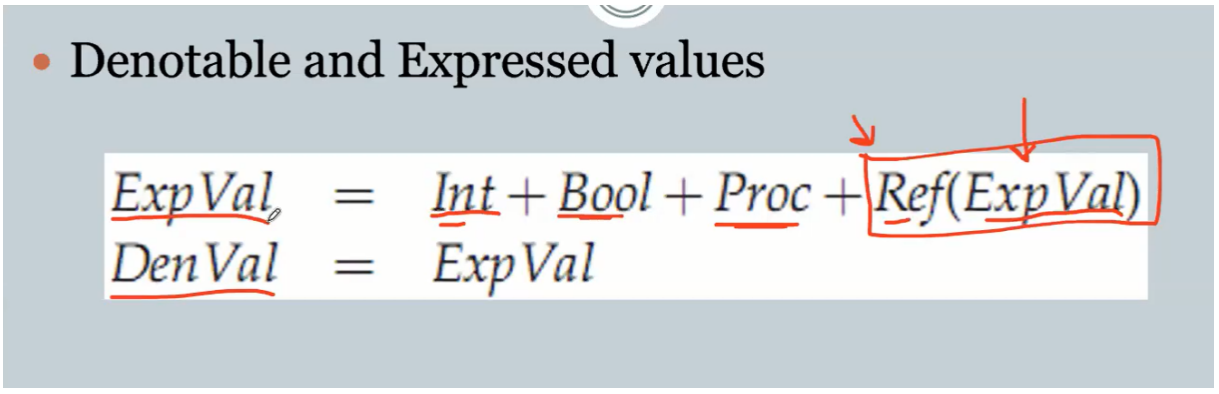

Language: letrec
Run
AST
ENV-STORE
Source Code
letrec f(x) = if zero?(x) then 0 else -((f -(x,1)), -2) in (f 4)
Output
#(struct:num-val 5)
#(struct:extend-env x #(struct:num-val 10) #(struct:empty-env))))))
(num-val 8)
letrec-exp
p-name
b-var
p-body
letrec-body
f
x
if-exp
call-exp
exp1
exp2
exp3
rator
rand
zero?-exp
const-exp
diff-exp
var-exp
const-exp
num
var
0
call-exp
const-exp
f
4
rator
rand
num
-2
exp1
exp2
f
var-exp
const-exp
1

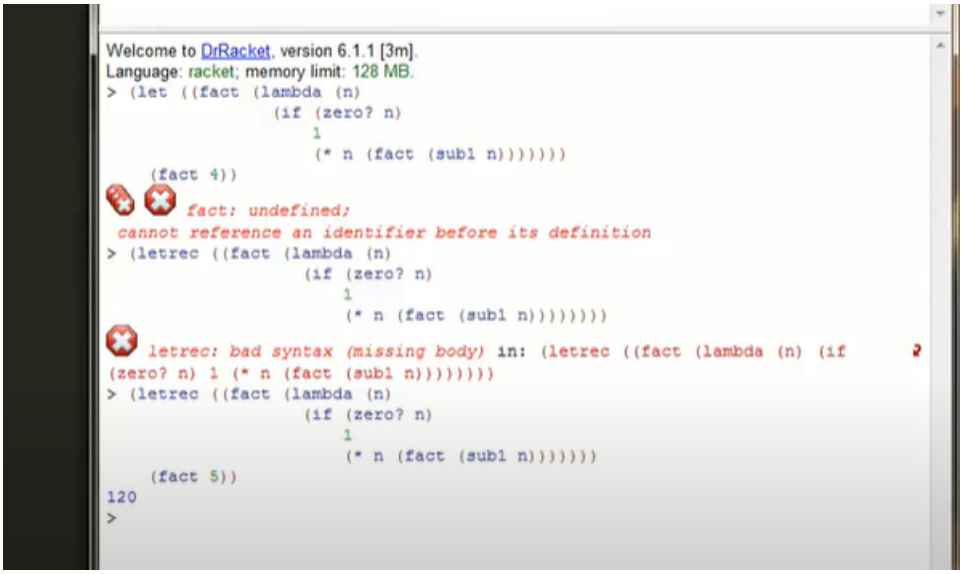

Source Code
letrec f(x) = if zero?(x) then 0 else -((f -(x,1)), -2) in (f 4)
Output
#(struct:num-val 5)
#(struct:extend-env x #(struct:num-val 10) #(struct:empty-env))))))
(num-val 8)
Environment and Store Tables
Current Expression
(var-exp 'f)
Environment
f
i
v
x
(if-exp (zero?-exp (var-exp 'x)) (const-exp 0) (diff-exp (call-exp (var-exp 'f) (diff-exp (var-exp 'x) (const-exp 1))) (const-exp -2)))
(num-val 1)
(num-val 5)
(num-val 10)

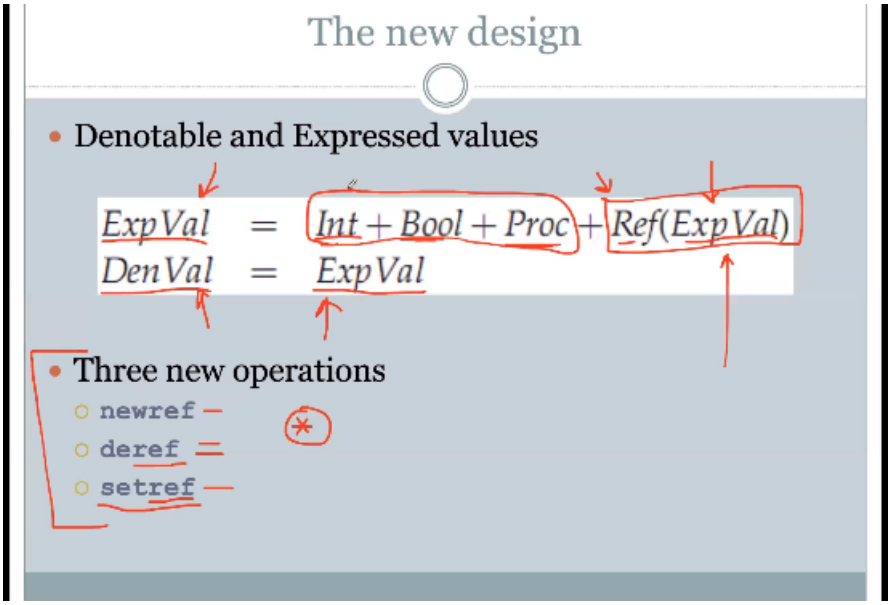


EREF VS IREF : U CAN STORE INFO

C VS JAVA POINTER STORAGE



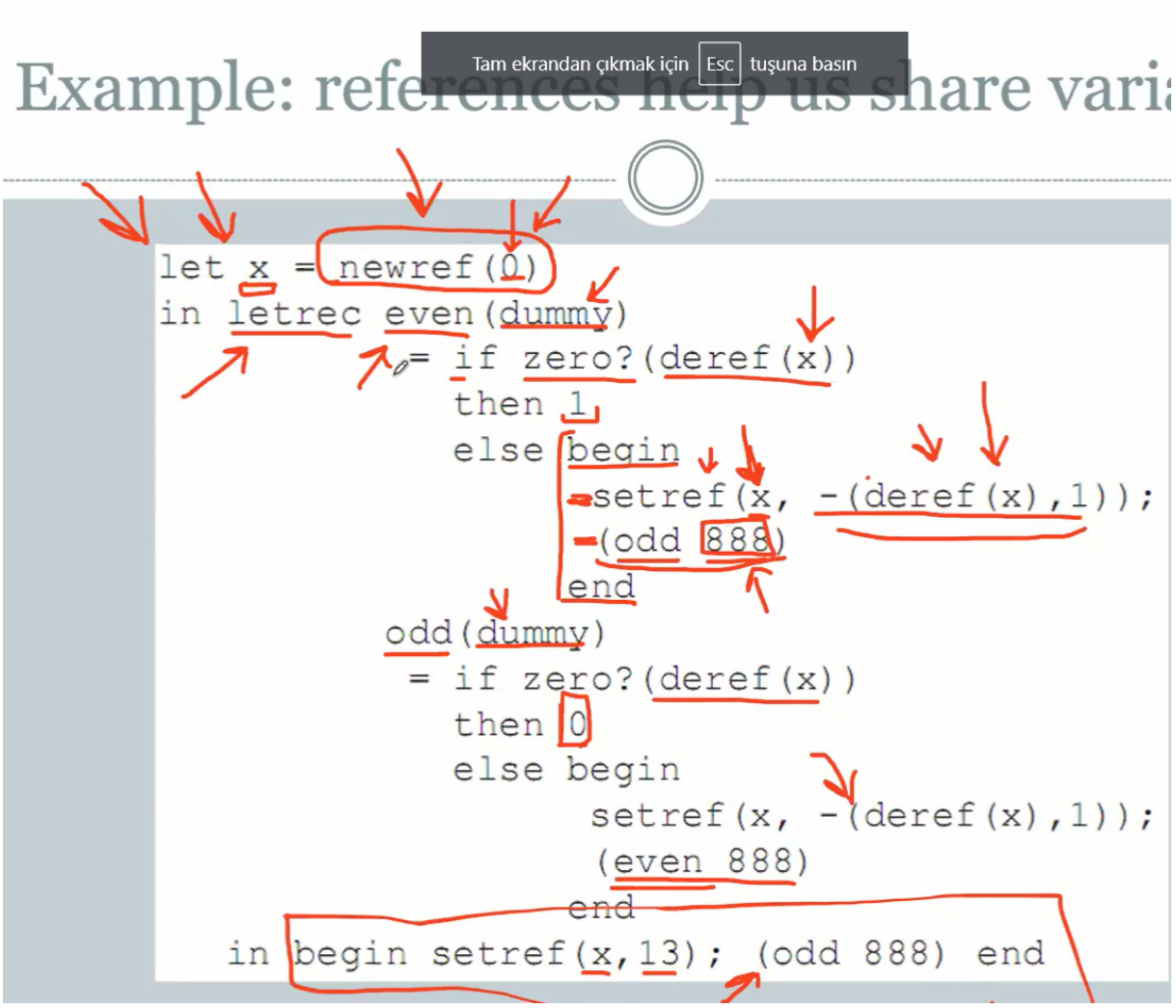




Let x = NEWREF(0): set x equal to a location where its value = 0

DEREF(x): return the value located at the address = x

SETREF(x,13): set the value located at address x to 13



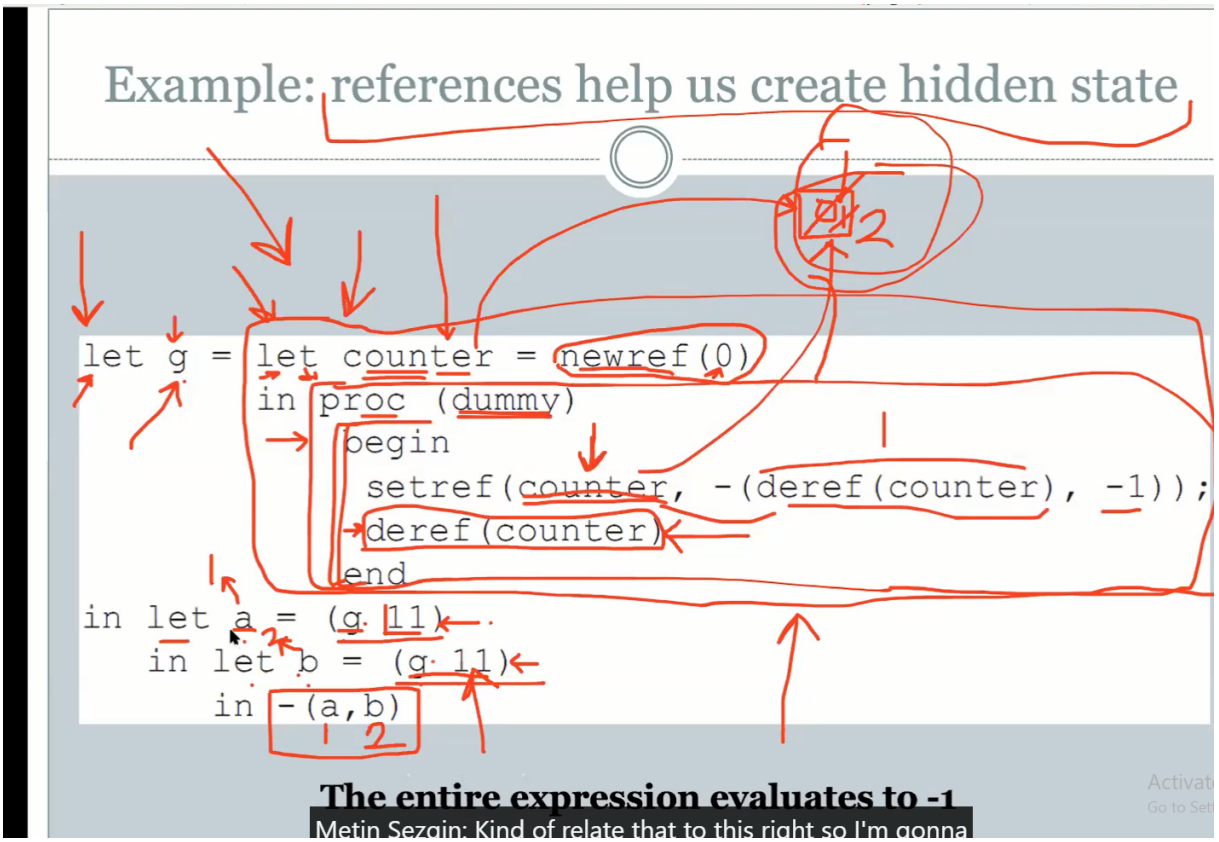
X= even will return if Even() called 1

X= even will return if odd() called 0

X= odd will return if Even() called 0

X= odd will return if odd() called 1

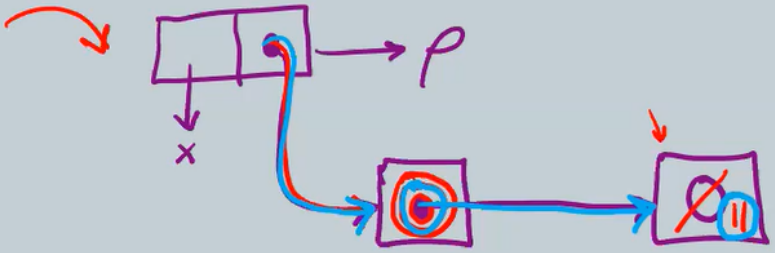
Example: references help us create hidden state

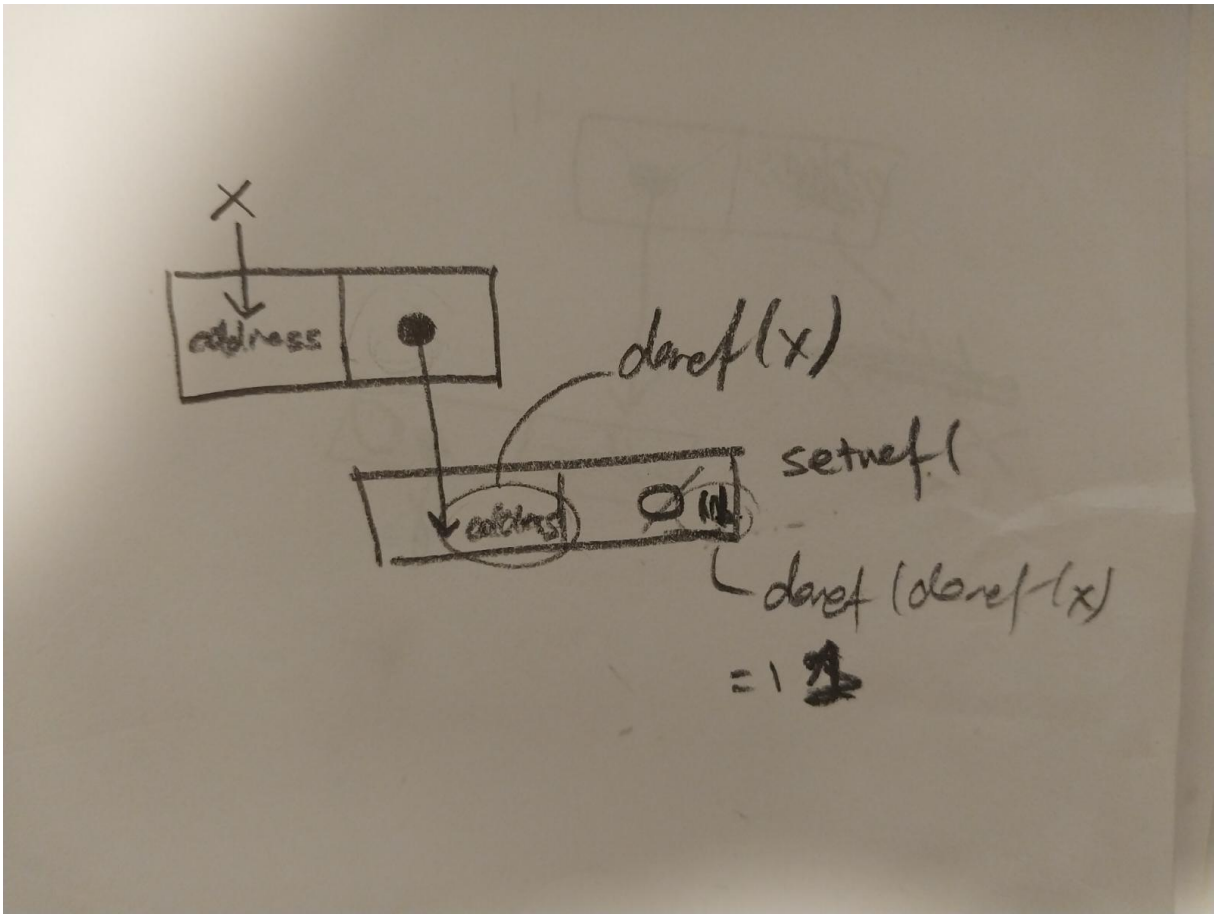


The entire expression evaluates to -1

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

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





Store passing specifications

- The new **value-of** $(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (val_1, \sigma_1)$
- Example $(\texttt{value-of}\ (\texttt{const-exp}\ n)\ \rho\ \sigma) = (n, \sigma)$
- More examples

$$\frac{(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (val_1, \sigma_1) \quad (\texttt{value-of}\ exp_2\ \rho\ \sigma_1) = (val_2, \sigma_2)}{(\texttt{value-of}\ (\texttt{diff-exp}\ exp_1\ exp_2)\ \rho\ \sigma_0) = (\lceil \lfloor val_1 \rfloor - \lfloor val_2 \rfloor \rceil, \sigma_2)}$$

$$\frac{(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (val_1, \sigma_1)}{(\texttt{value-of}\ (\texttt{if-exp}\ exp_1\ exp_2\ exp_3)\ \rho\ \sigma_0) = \begin{cases} (\texttt{value-of}\ exp_2\ \rho\ \sigma_1) & \text{if } (\texttt{expval->bool}\ val_1) = \#\texttt{t} \\ (\texttt{value-of}\ exp_3\ \rho\ \sigma_1) & \text{if } (\texttt{expval->bool}\ val_1) = \#\texttt{f} \end{cases}$$

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

$$\frac{\texttt{(value-of } exp\ \rho\ \sigma_0\texttt{)} = (val, \sigma_1) \qquad l \notin \mathrm{dom}(\sigma_1)}{\texttt{(value-of (newref-exp } exp\texttt{)}\ \rho\ \sigma_0\texttt{)} = (\texttt{(ref-val } l\texttt{)}, [l{=}val]\sigma_1)}$$

$$\frac{\texttt{(value-of } exp\ \rho\ \sigma_0\texttt{)} = (l, \sigma_1)}{\texttt{(value-of (deref-exp } exp\texttt{)}\ \rho\ \sigma_0\texttt{)} = (\sigma_1(l), \sigma_1)}$$

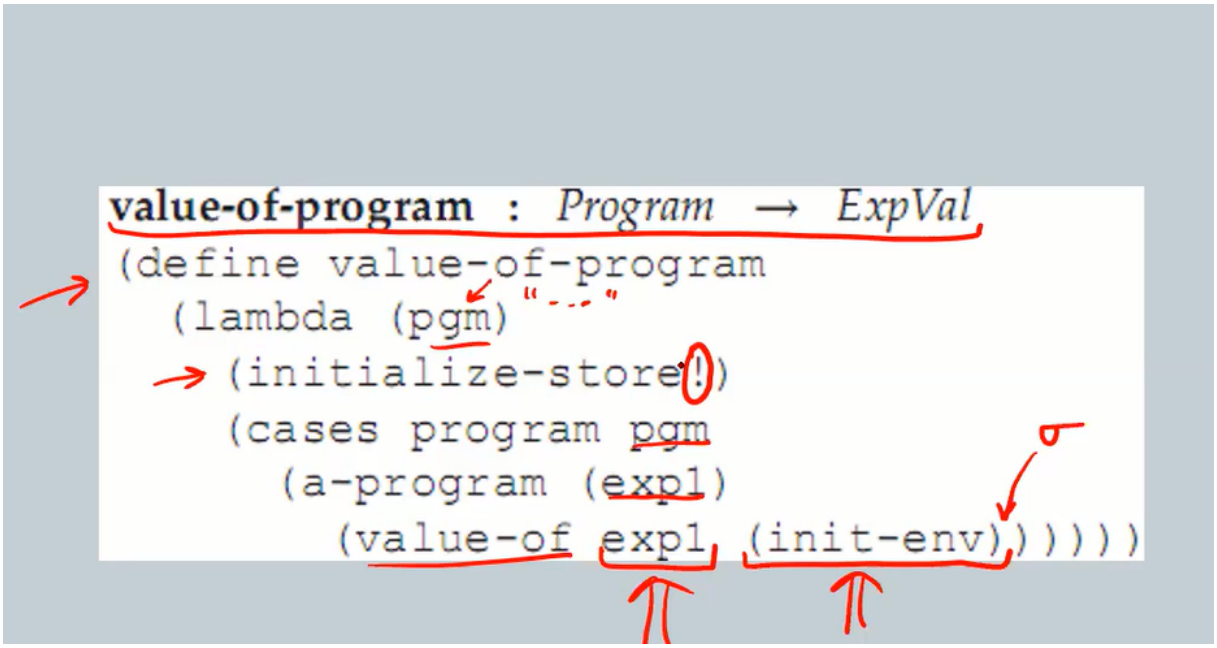
$$\frac{\texttt{(value-of } exp_1\ \rho\ \sigma_0\texttt{)} = (l, \sigma_1) \qquad \texttt{(value-of } exp_2\ \rho\ \sigma_1\texttt{)} = (val, \sigma_2)}{\texttt{(value-of (setref-exp } exp_1\ exp_2\texttt{)}\ \rho\ \sigma_0\texttt{)} = (23, [l{=}val]\sigma_2)}$$

a new address

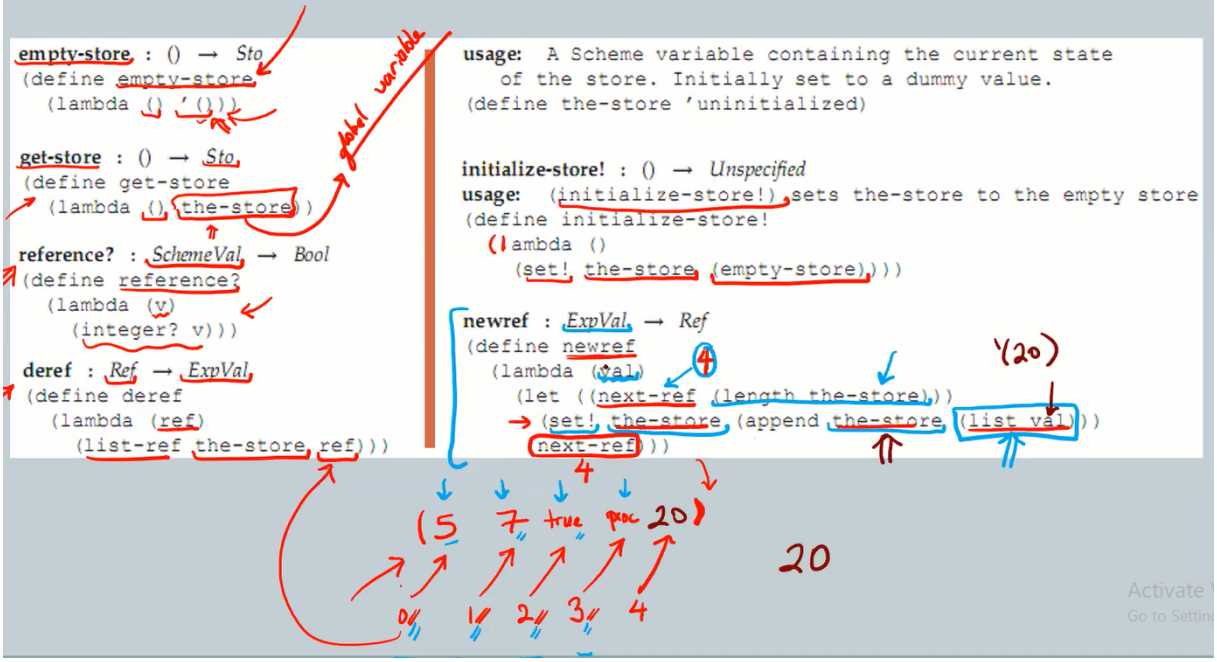
return the contents of location l

-(5,2)

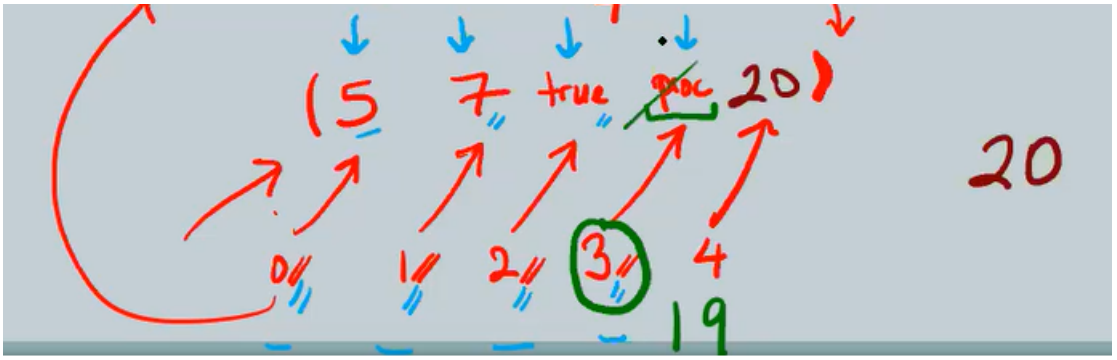
3



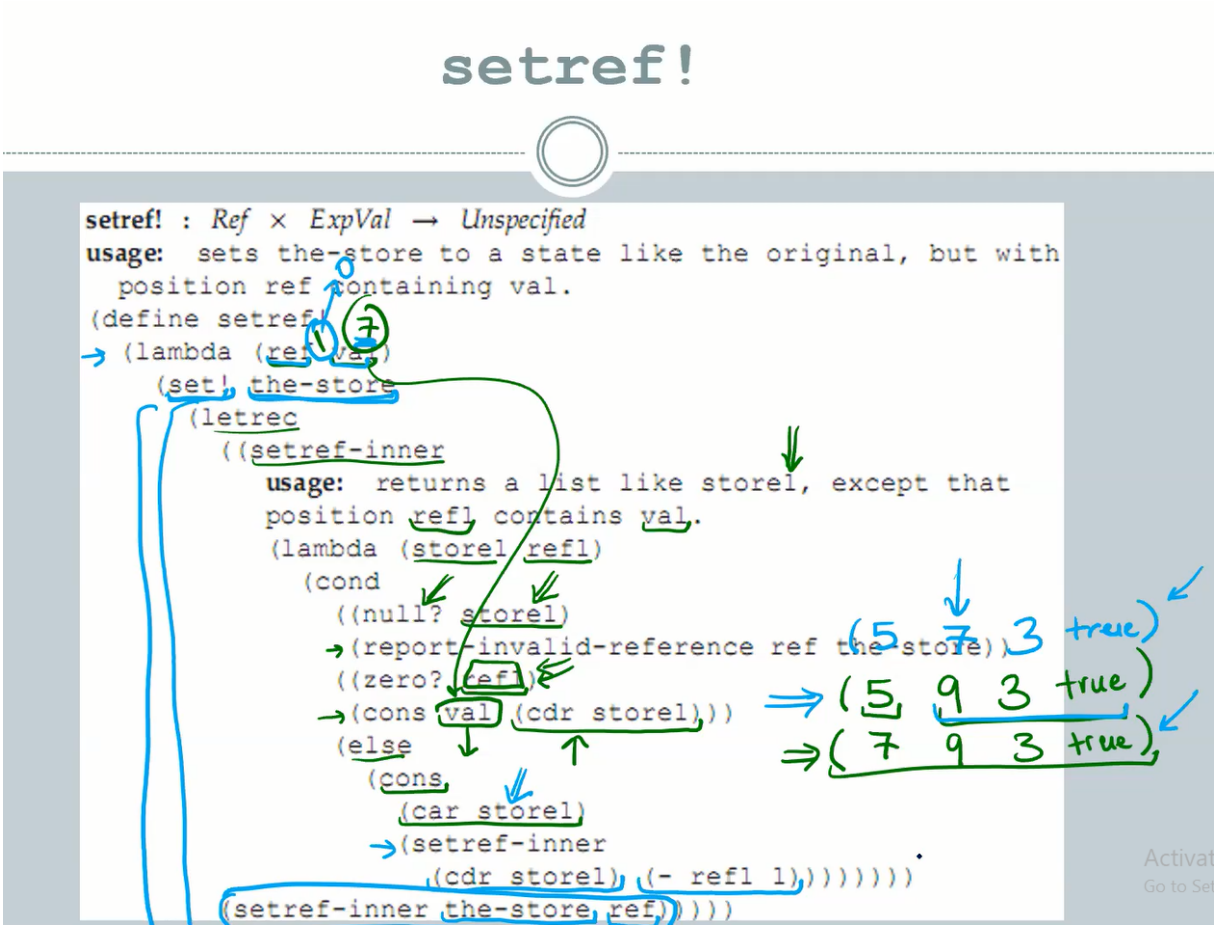
Implementation of Stores



Implementation of setref:



Lets say you want to set the value at locaiton 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 itgets to that location it puts the 19 and cons the remainder or rest of the list which in this case is 20.

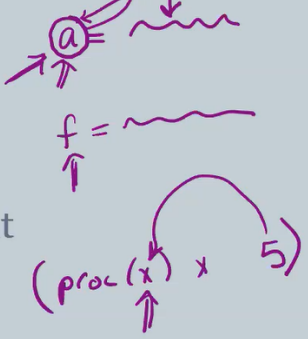


Implementation
newref-exp, deref-exp, setref-exp

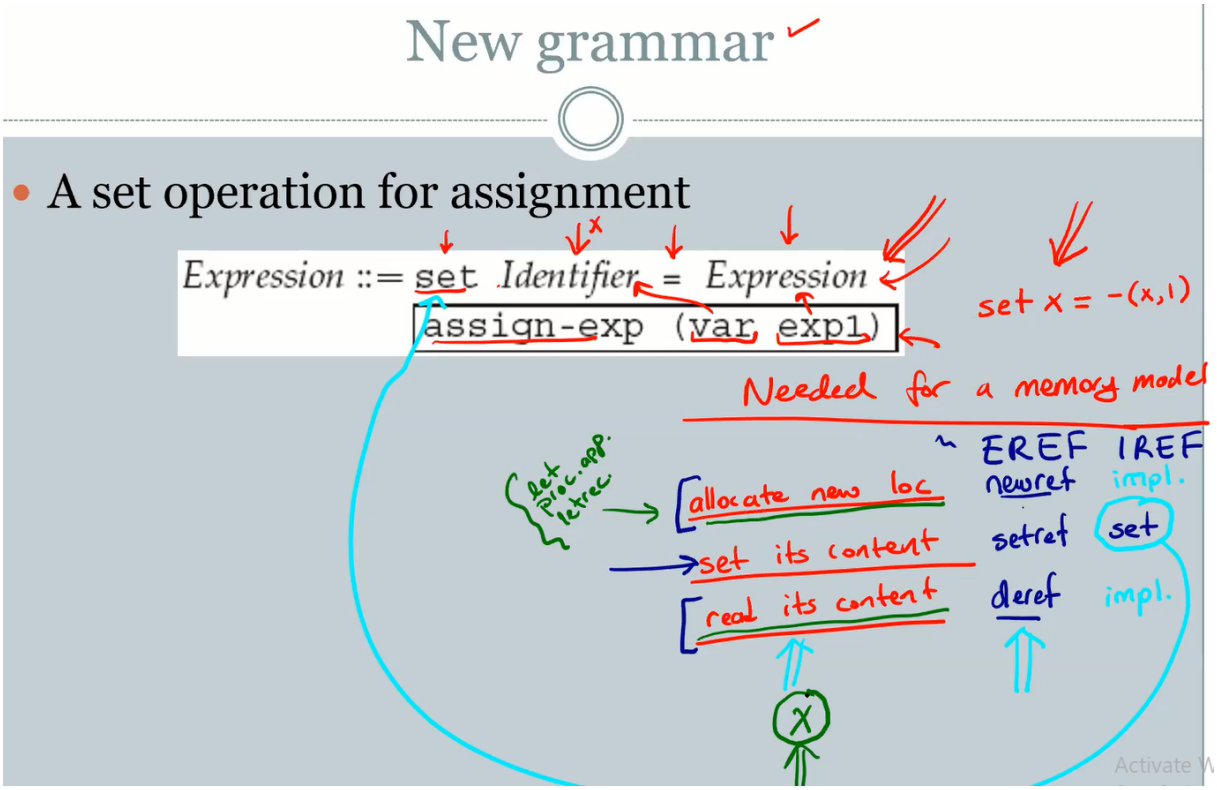
```
(newref-exp (exp1)
  (let ((v1 (value-of exp1 env)))
    (ref-val (newref v1))))

(deref-exp (exp1)
  (let ((v1 (value-of exp1 env)))
    (let ((ref1 (expval->ref v1)))
      (deref ref1))))

(setref-exp (exp1 exp2)
  (let ((ref (expval->ref (value-of exp1 env))))
    (let ((val2 (value-of exp2 env)))
      (begin
        (setref! ref val2)
        (num-val 23)))))
```

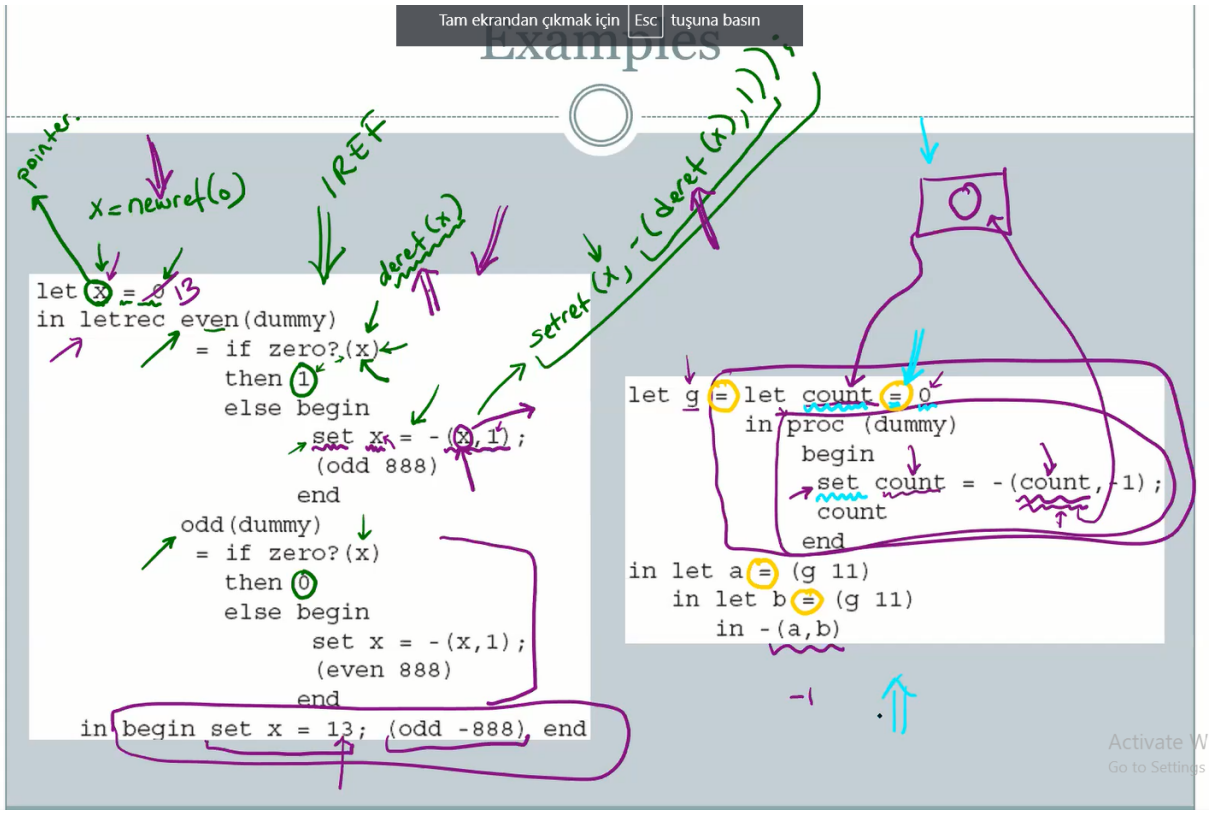

Implicit references

- 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

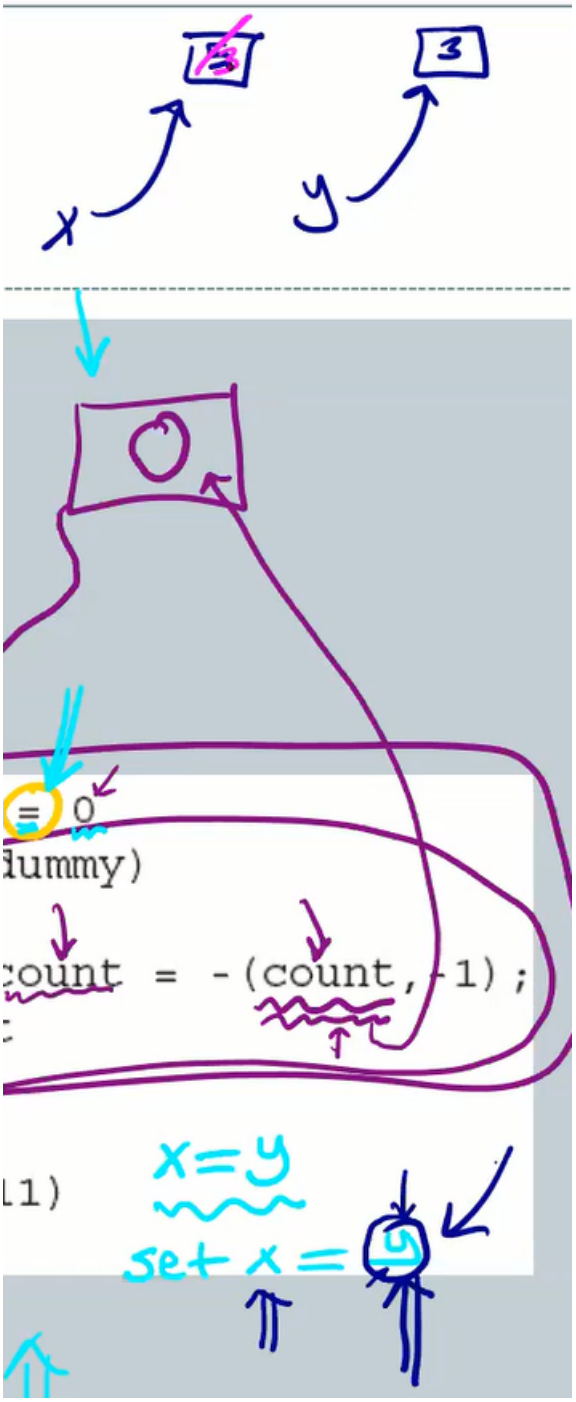


$$\begin{aligned} ExpVal &= Int + Bool + Proc \\ DenVal &= Ref(ExpVal) \end{aligned}$$





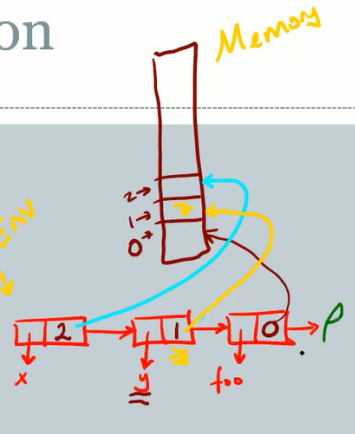
IREF cannot make references to references unlike EREF


3
x
y
= 0
dummy)
count = -(count, -1);
x=y
set x =

Behavior specification

- **var-exp**

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

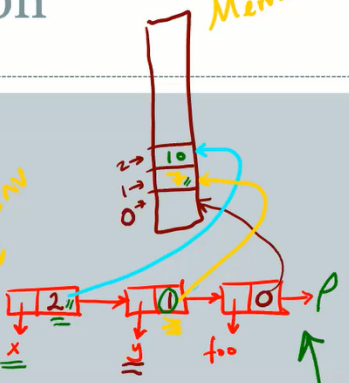


let foo = ~ in
let y = 7 in
let x = ~ in
y

Behavior specification

- **var-exp**

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



- **assign-exp**

$$\frac{(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (val_1, \sigma_1)}{(\texttt{value-of}\ (\texttt{assign-exp}\ var\ exp_1)\ \rho\ \sigma_0) = (\lceil 27 \rceil, [\rho(var) = val_1]\sigma_1)}$$

- **apply-procedure**

let foo = ~
let y = 1 in
let x = 2 in
set x = y

Behavior specification

var-exp

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

assign-exp

$$\frac{(\text{value-of}\ exp_1\ \rho\ \sigma_0) = (val_1, \sigma_1)}{(\text{value-of}\ (\text{assign-exp}\ var\ exp_1)\ \rho\ \sigma_0) = (\lceil 27 \rceil, [\rho(var) = val_1]\sigma_1)}$$

apply-procedure

$$(\text{apply-procedure}\ (\text{procedure}\ var\ body\ \rho)\ val\ \sigma) = (\text{value-of}\ body\ [var = l]\rho\ [l = val]\sigma)$$

Implementation

- **var-exp**

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

- **assign-exp**

```
(assign-exp (var exp1)
  (begin
    (setref!
      (apply-env env var)
      (value-of exp1 env))
    (num-val 27)))
```

- **apply-procedure**

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 (newref val) saved-env))))))
```

Implementation

Reference instantiations

- 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))))
```

IREF

In addition we want mutation

- New grammar

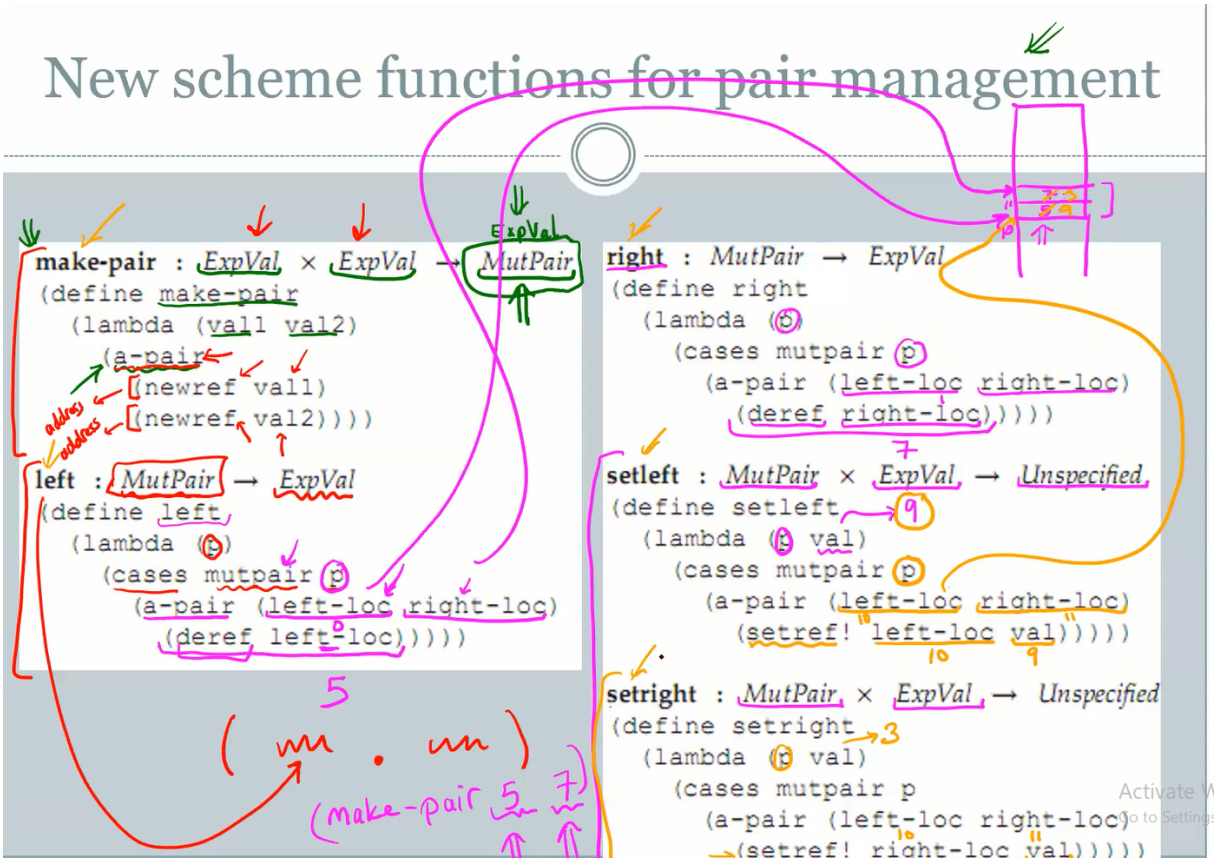
```
newpair   : Expval × Expval → MutPair
left      : MutPair → Expval
right     : MutPair → Expval
setleft   : MutPair × Expval → Unspecified
setright  : MutPair × Expval → Unspecified
```

- New set of
  - Denotables
  - Expressibles

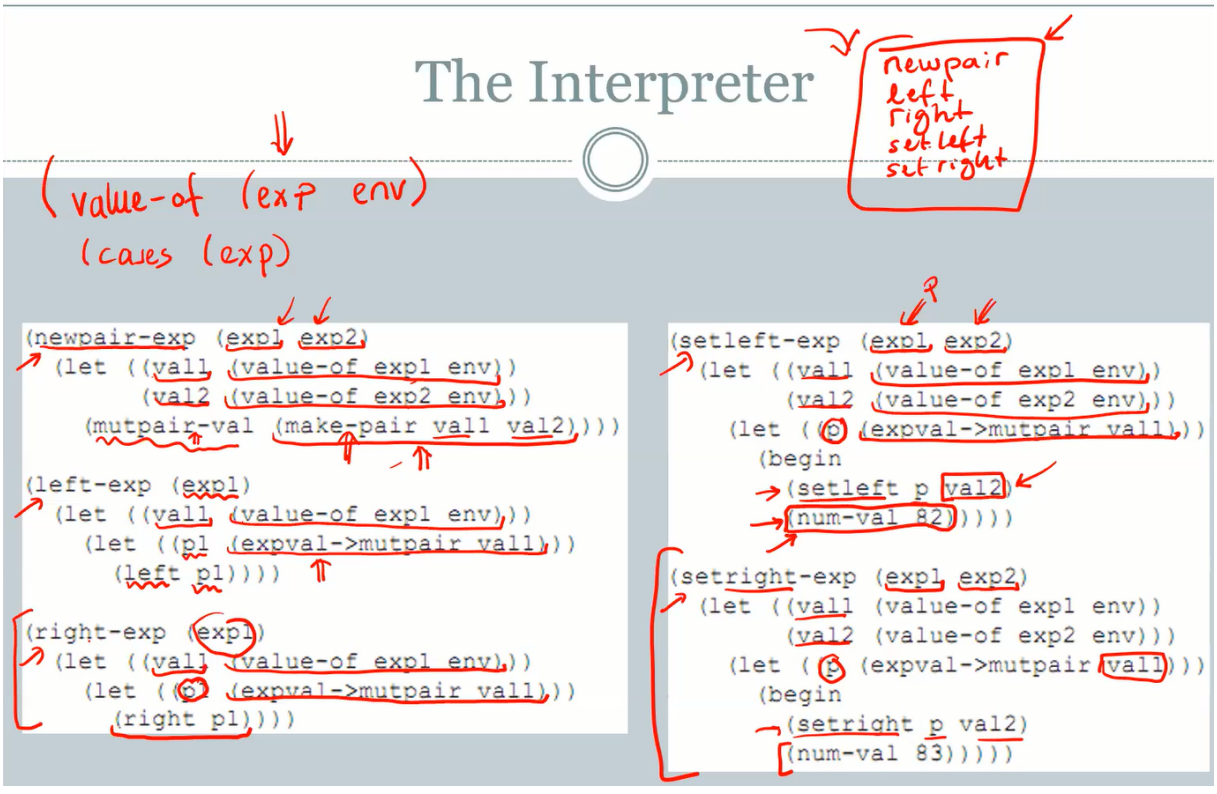
$$
\begin{aligned}
ExpVal &= Int + Bool + Proc + MutPair \\
DenVal &= Ref(ExpVal) \\
MutPair &= Ref(ExpVal) \times Ref(ExpVal)
\end{aligned}
$$

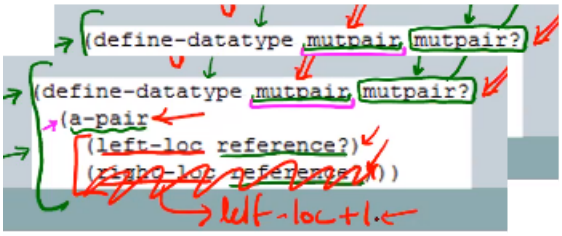
```
(define-datatype expval expval?
  (num-val
    (value number?))
  (bool-val
    (boolean boolean?))
  (proc-val
    (proc proc?))
  (mutpair-val
    (p mutpair?))
  )
```

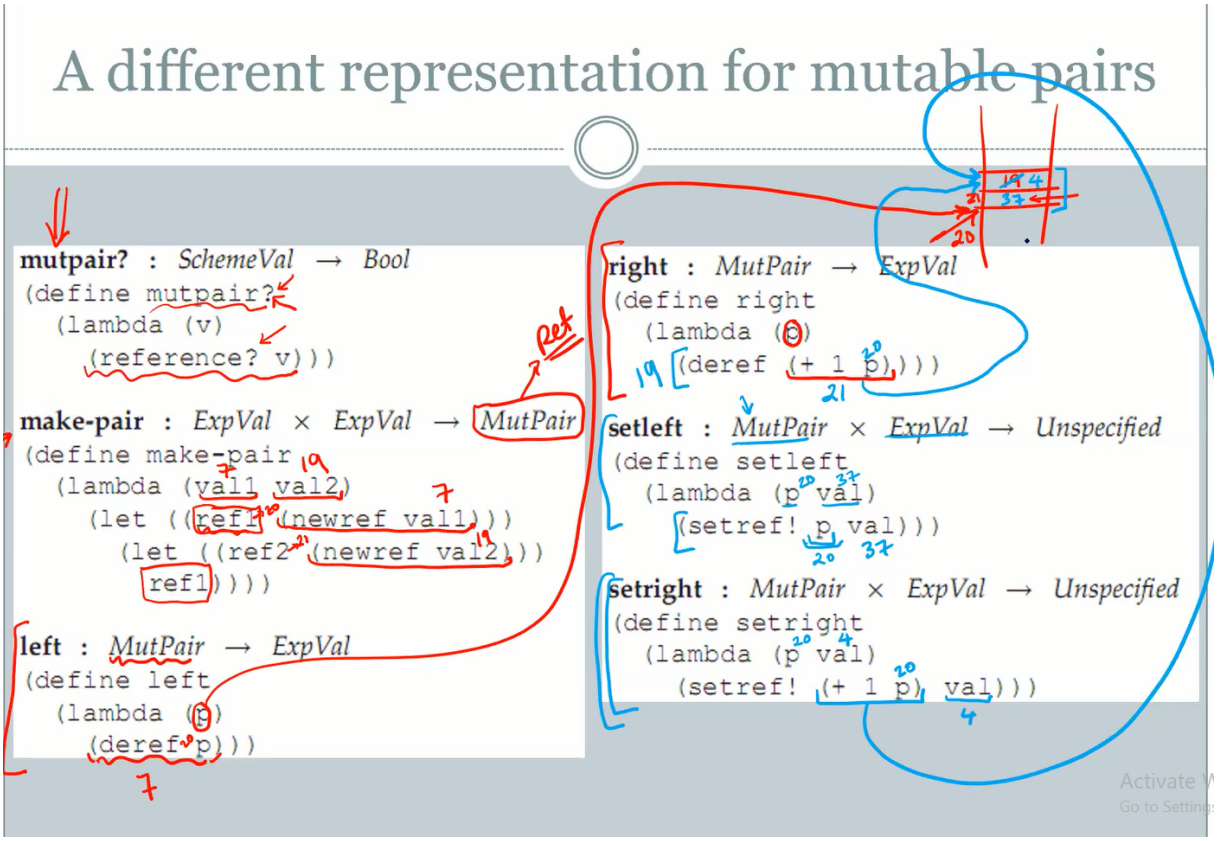
```
(define-datatype mutpair mutpair?
  (a-pair
    (left-loc reference?)
    (right-loc reference?)))
```



İnterpreter = valueof







- Alright, so how would you, for example, implement stacks. I would implement stacks by keeping

40:07

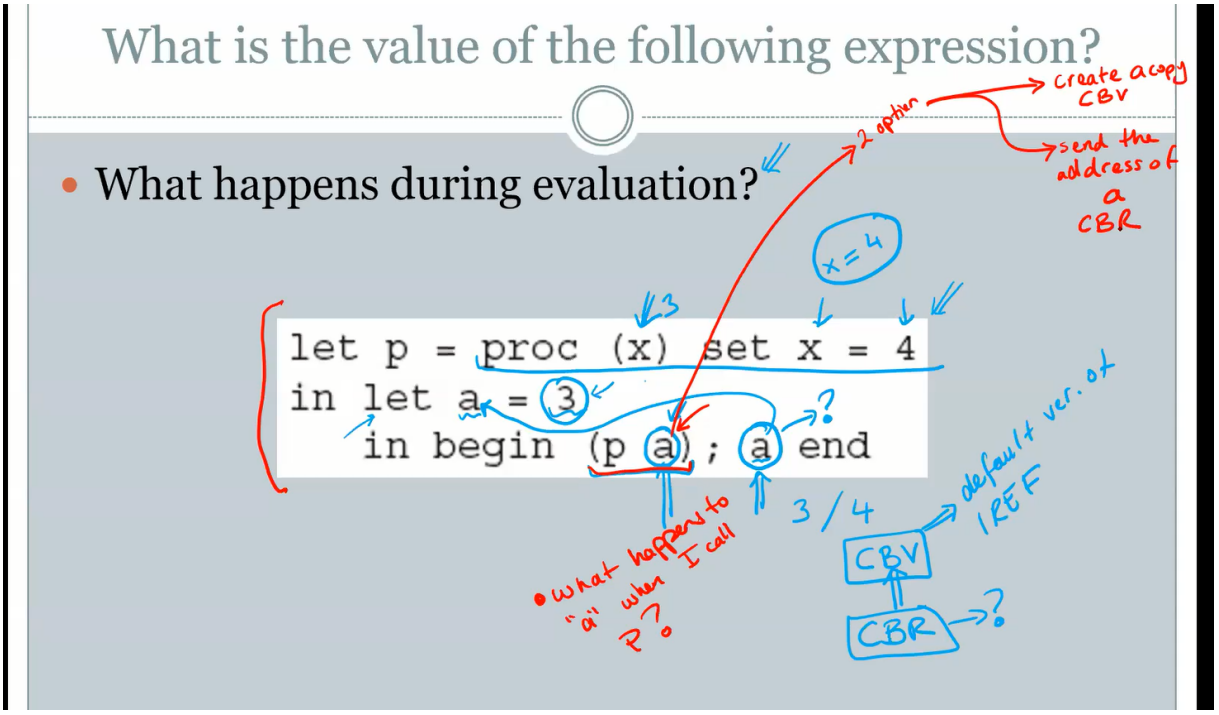
- Metin Sezgin: Track of the pointers that we would need in order to be able to push and pop things off the stack and the access to top value. If I'm implementing an array, how would I implement an array. I probably

40:20

- Metin Sezgin: Maybe store the beginning of the array in the memory, assuming that an array is really a sequence of contiguous

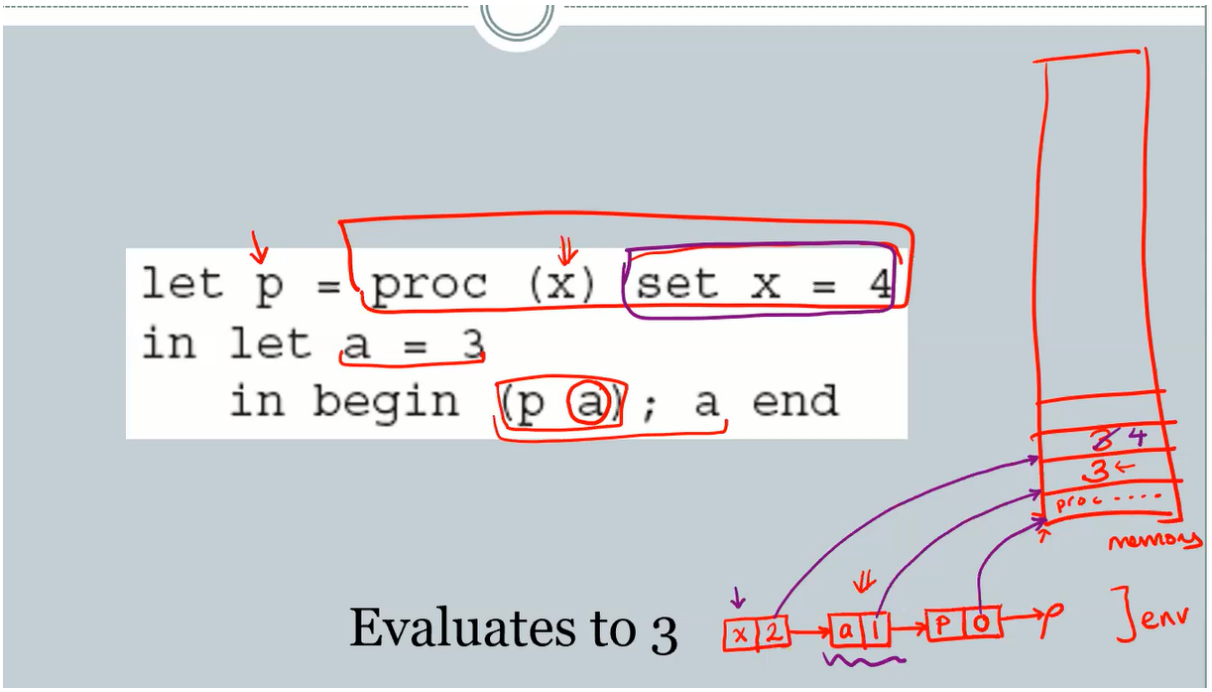
40:36

- 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

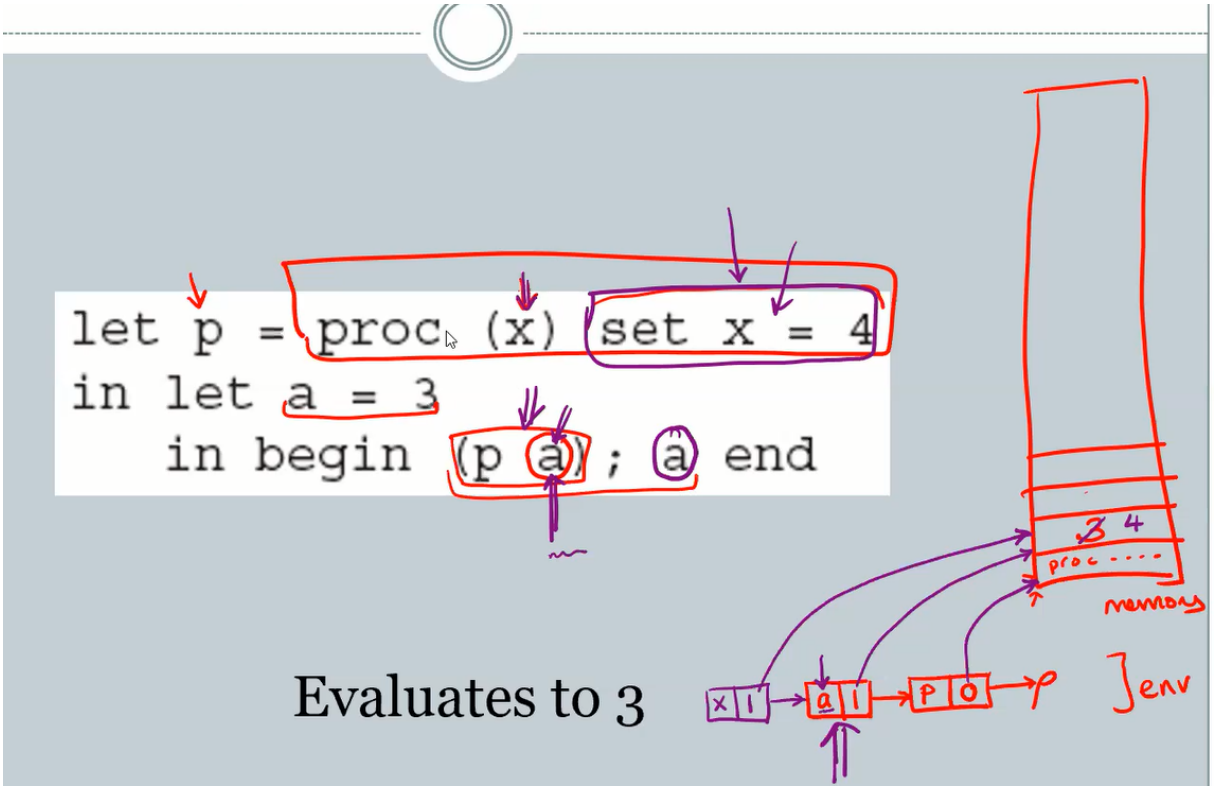


because default of iref is cbv or call by value we end up with a 3 output

IREF IN CBV:

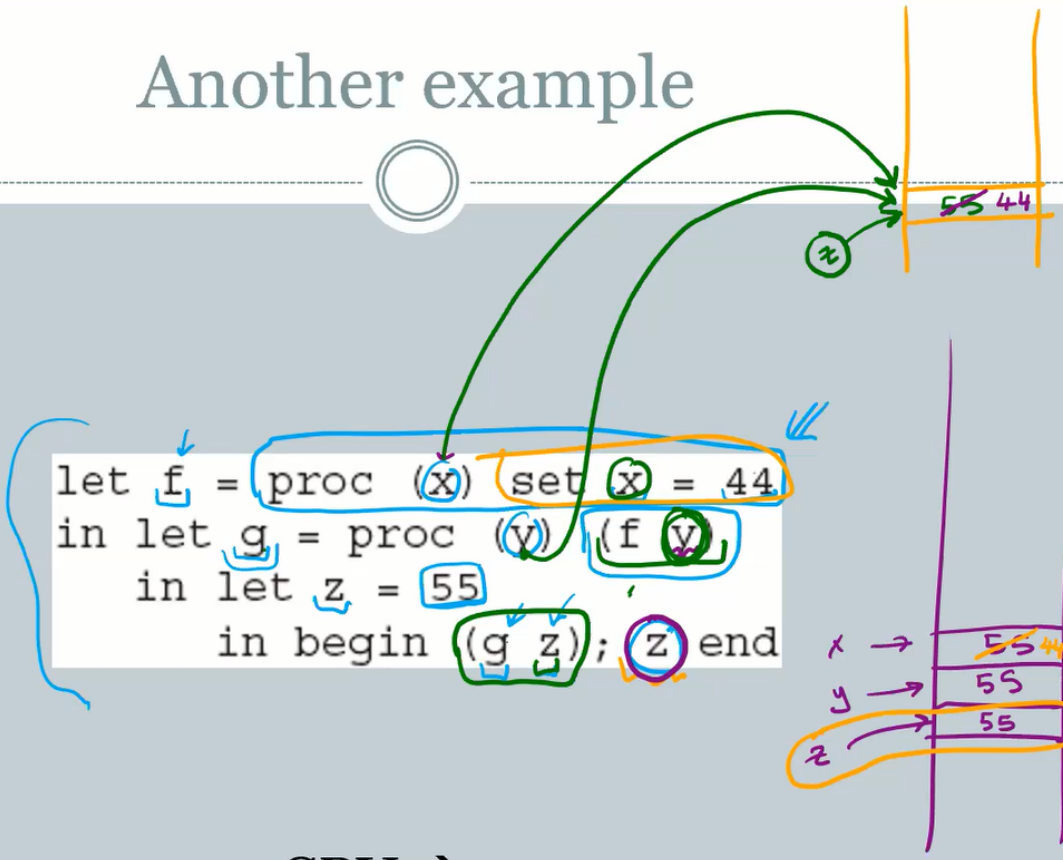


IREF IN CBR:

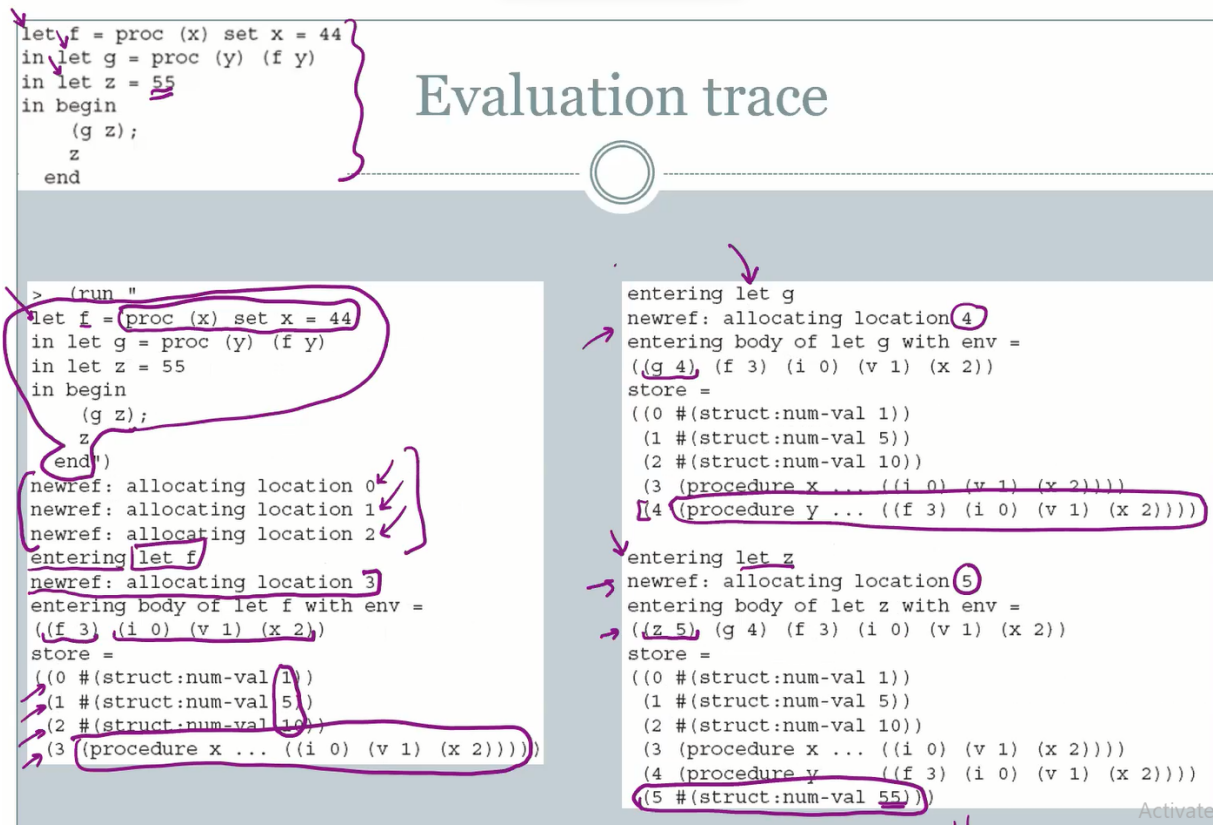


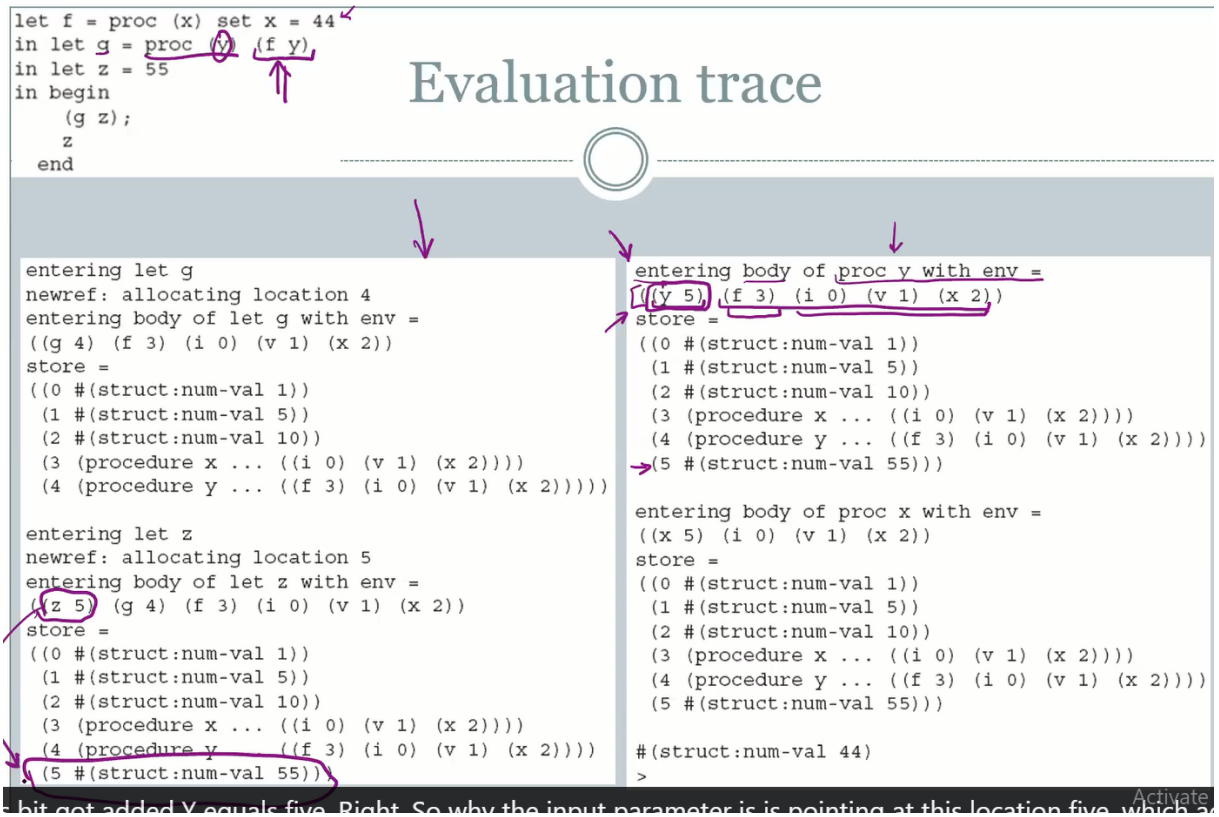
CBR: Swapping numbers with a function

Another example

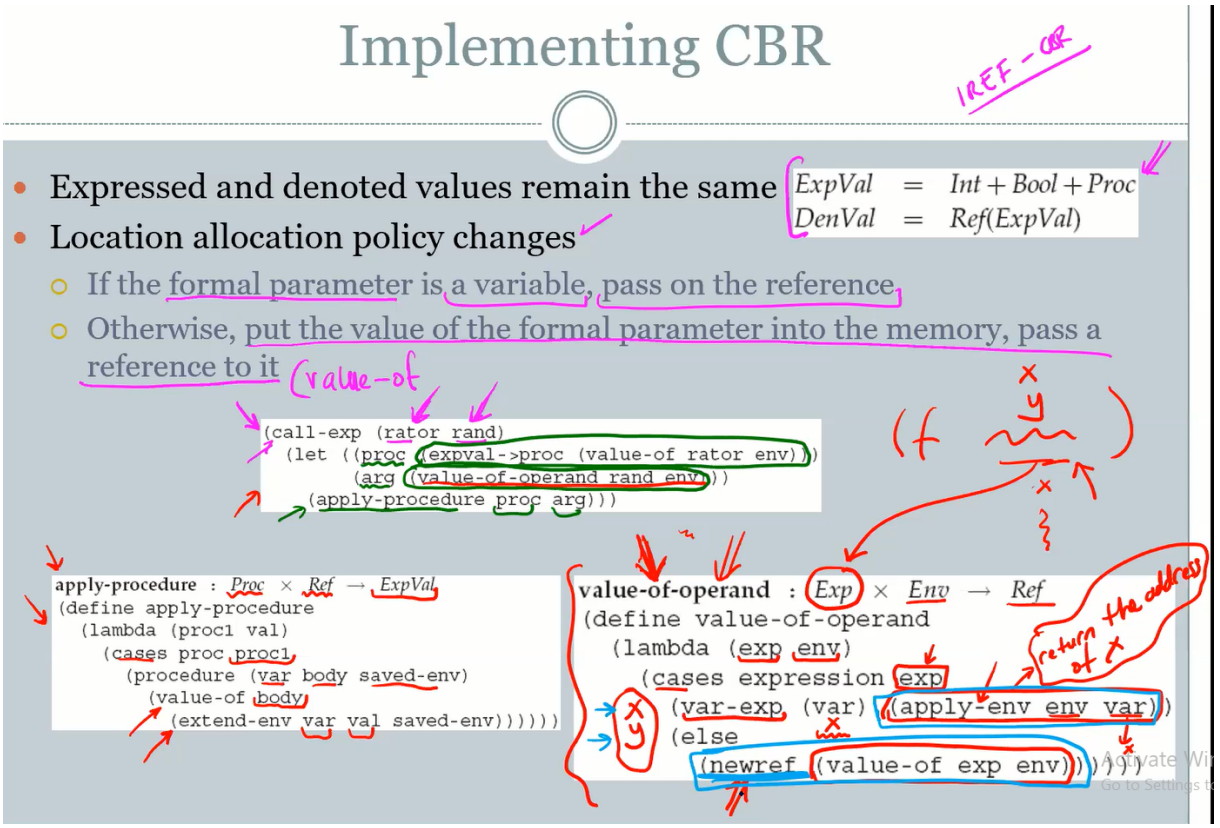


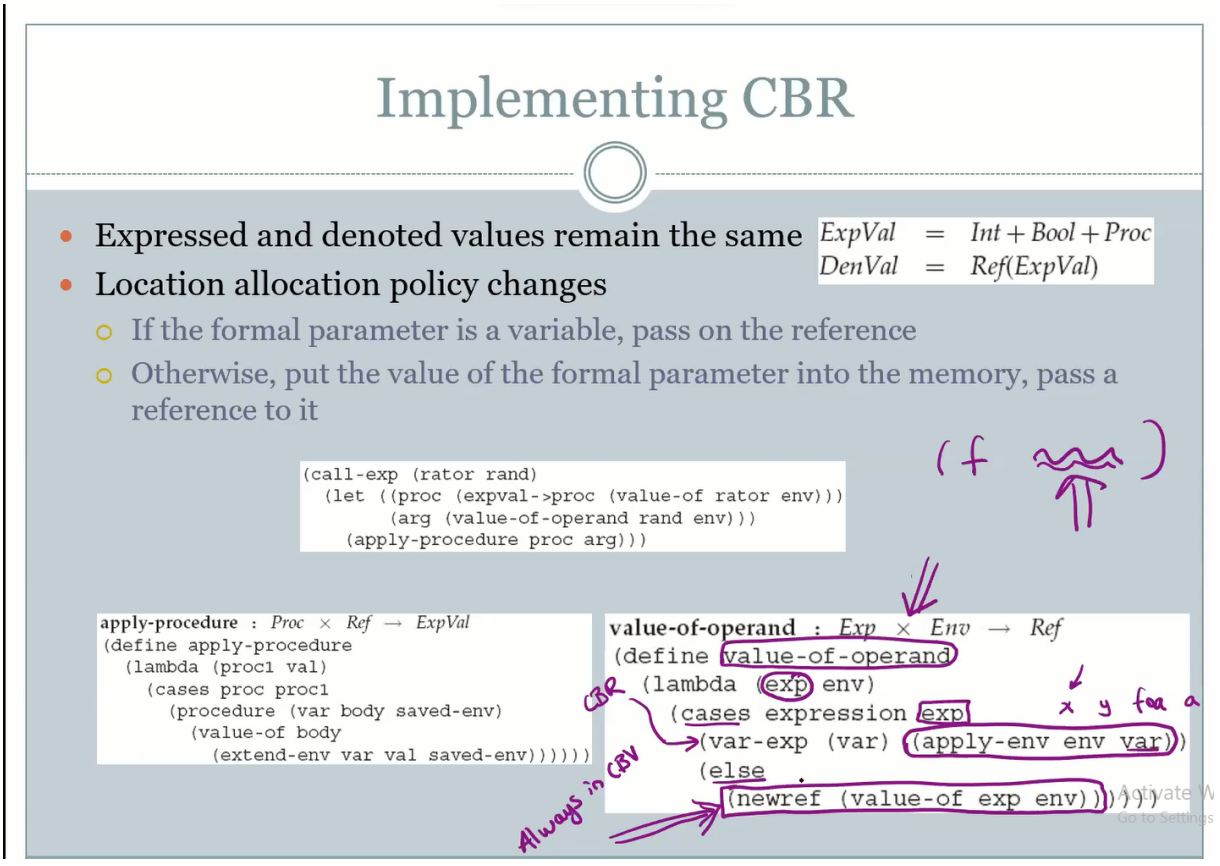
- CBV → 55
- CBR → 44





above is CBR case





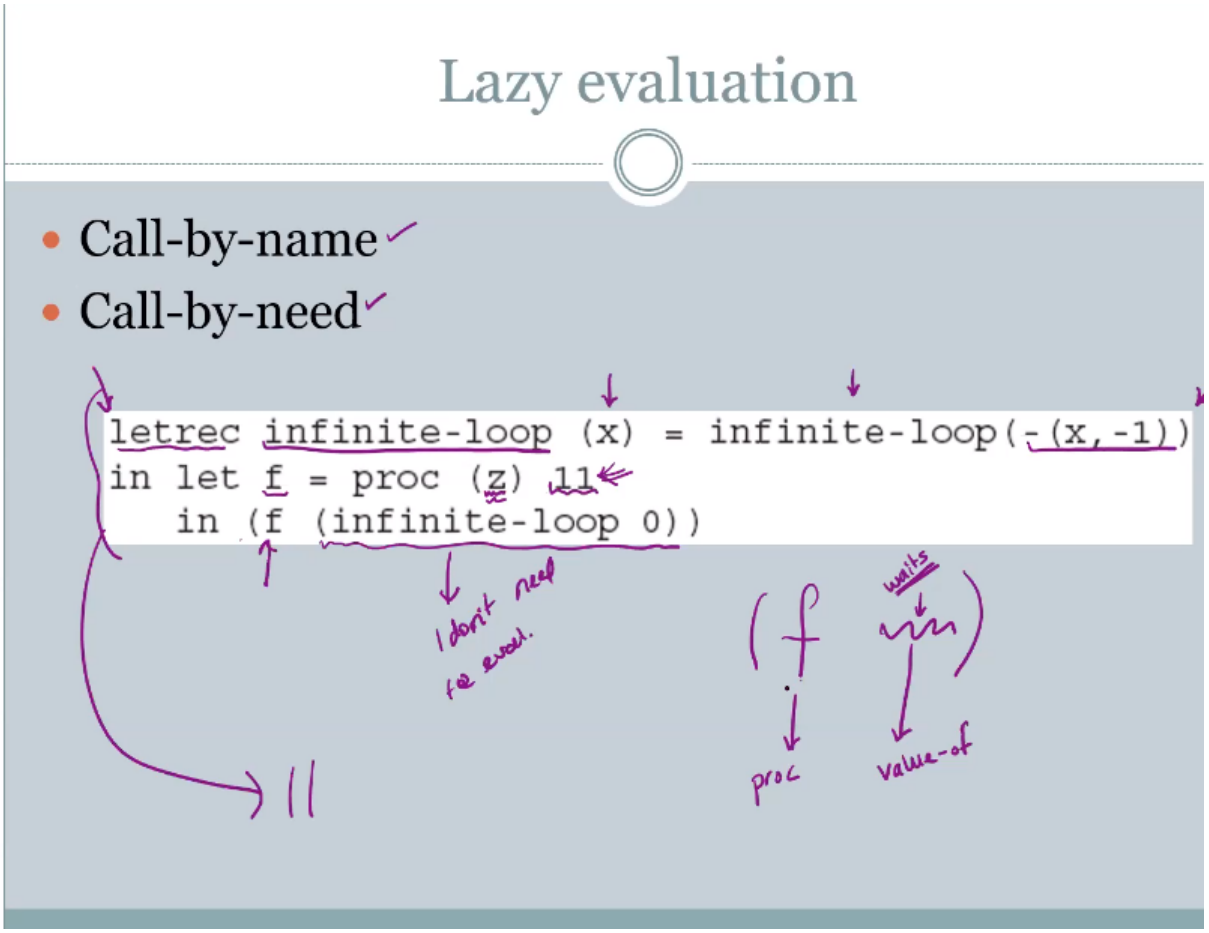
Above, when i CBR: if exp is a variable we know that it exists in the memory so we get te pointer tat points to that expression from the environment

Delet tte the line labelled wit CBR and ter u go u ave CBV

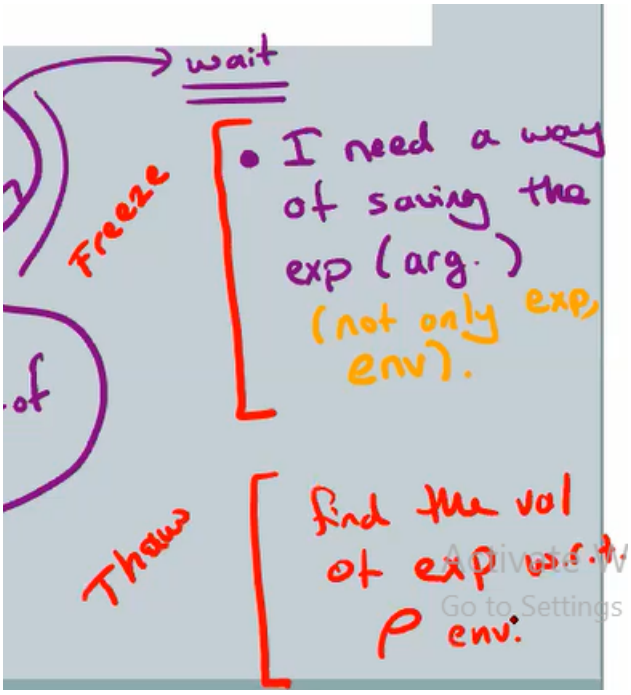
- lazy evaluation says let's wait

16:10

- 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

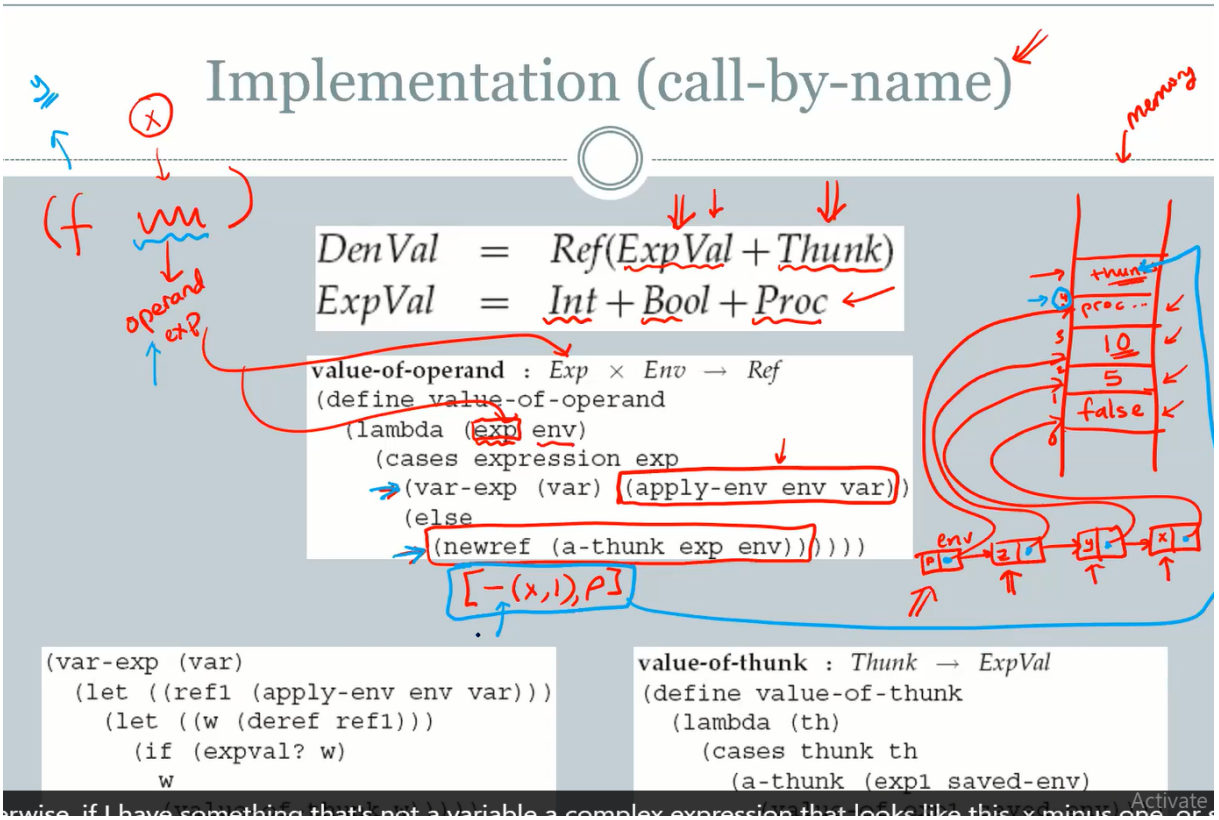


İ need to save the argument which waits since I might need to evaluate and use it later on



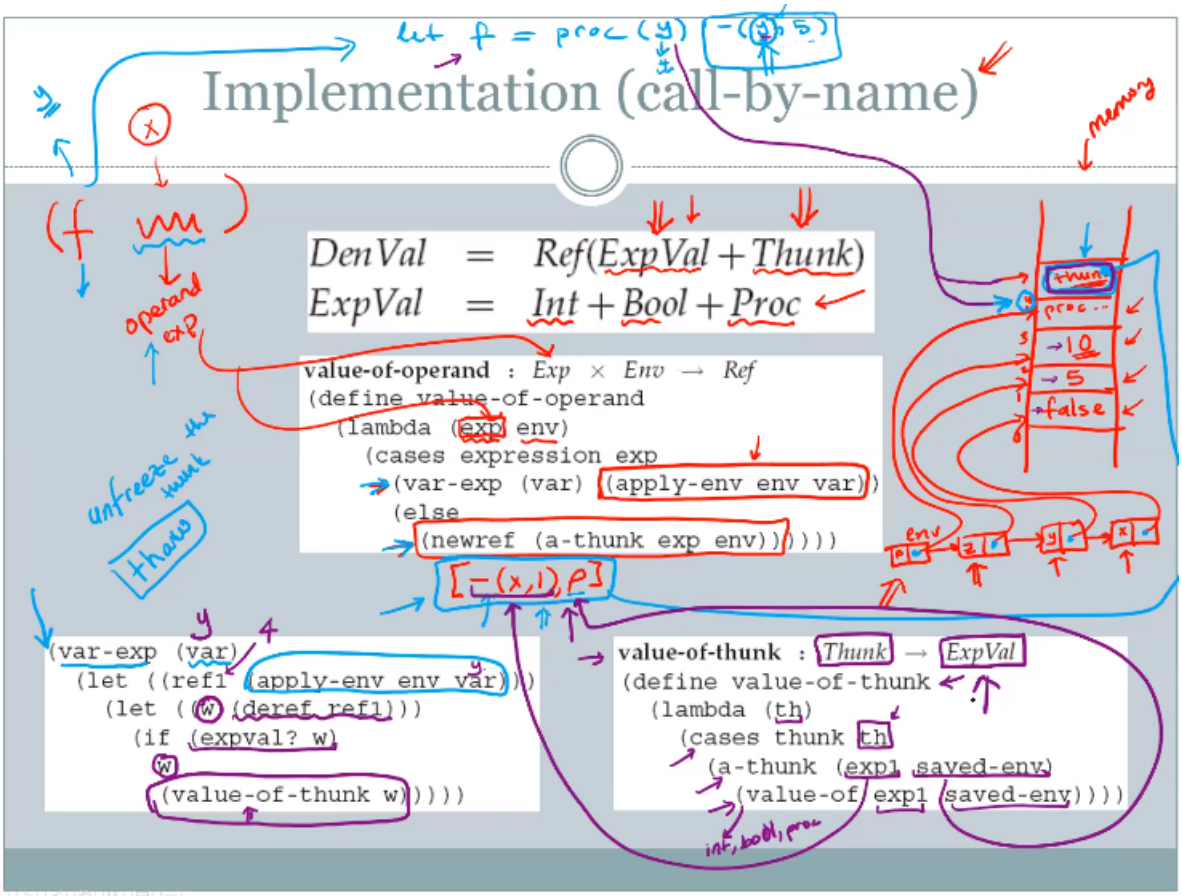

```
(define-datatype thunk thunk?
  (a-thunk
    (exp1 expression?)
    (env environment?)))
```

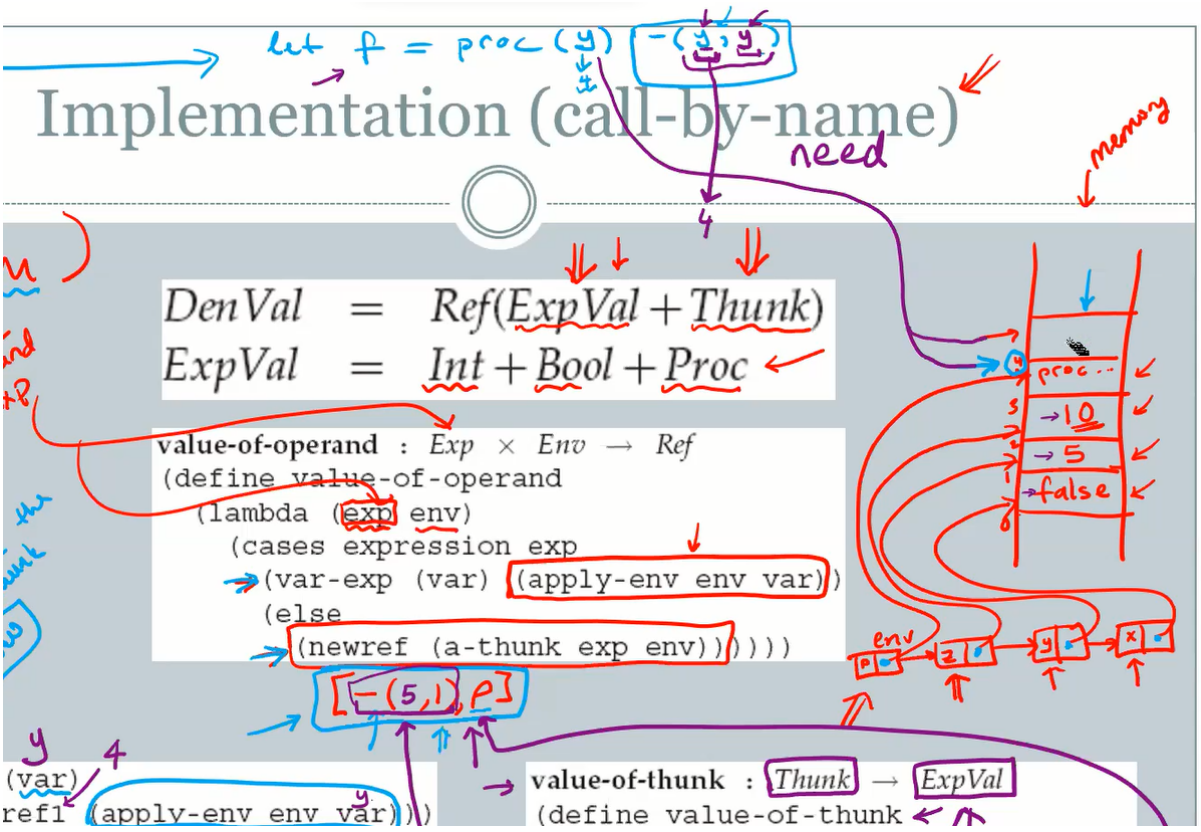
IREF EXTENSION: call by name



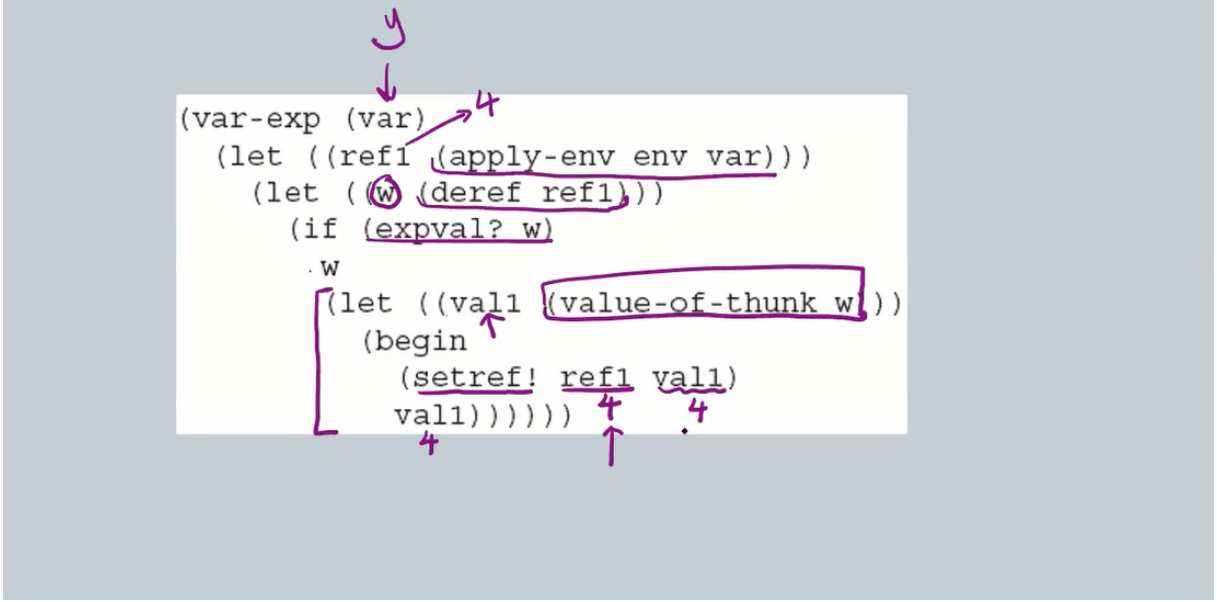
Here for valueofoperand if we r looking for a vlaue of a plain variable like x then we simply return its address location this is implemented ad the varexp var line otherwise if we have anything else we freeze the operation adn the environement and put it in an address

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


Implementation (call-by-name)
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))))))
(var-exp (var)
(let ((ref1 (apply-env env var)))
(let ((w (deref ref1)))
(if (expval? w)
w
(value-of-thunk w)))))
value-of-thunk : Thunk → ExpVal
(define value-of-thunk
(lambda (th)
(cases thunk th
(a-thunk (exp1 saved-env)
(value-of exp1 saved-env)))))
operand exp
unfreeze thunk
thaw
memory
env


Implementation (call-by-name)
need
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
memory
env

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
```

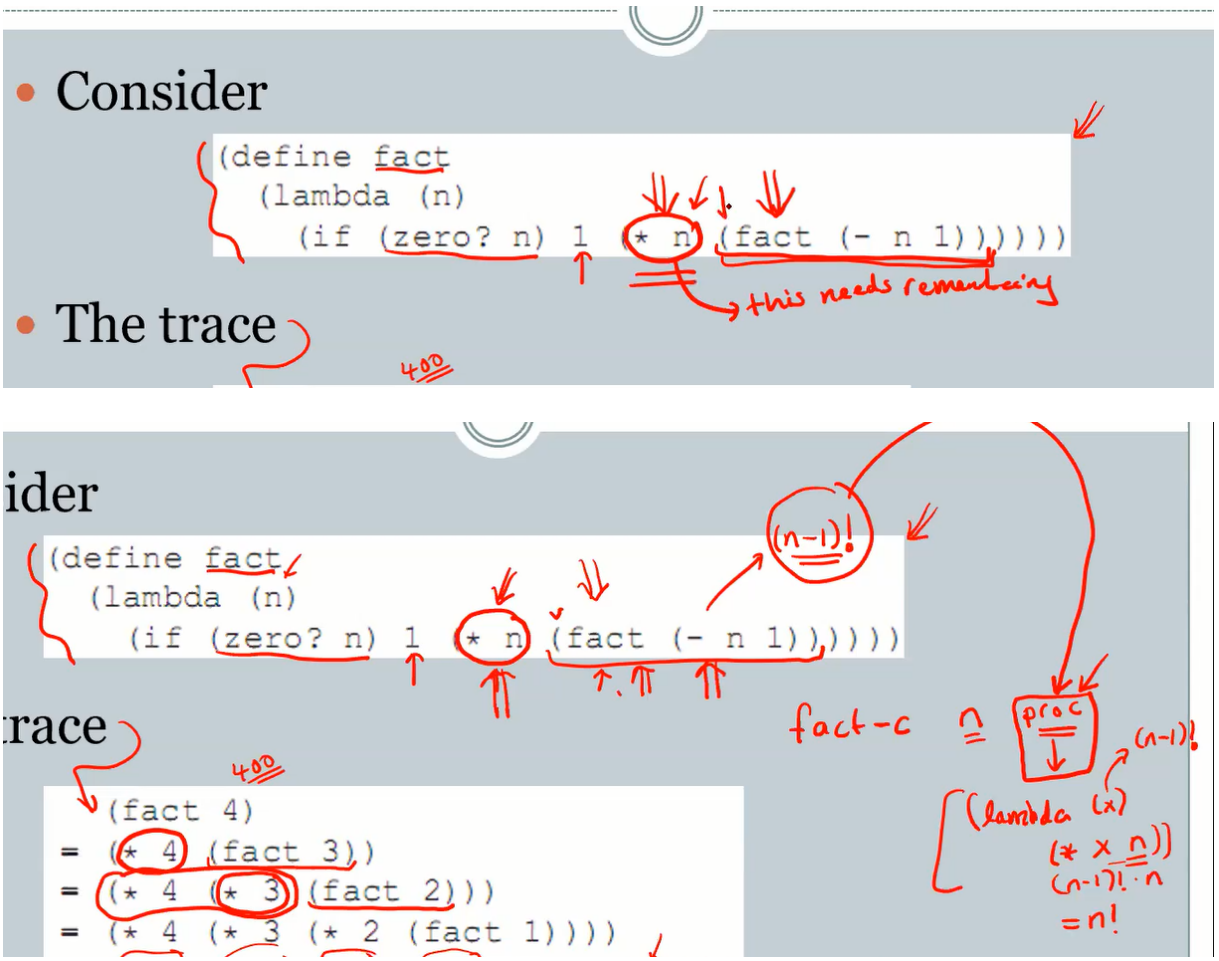

Recursive vs. Iterative Control Behavior

- 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
```



- the continuation of this procedure will know how to take

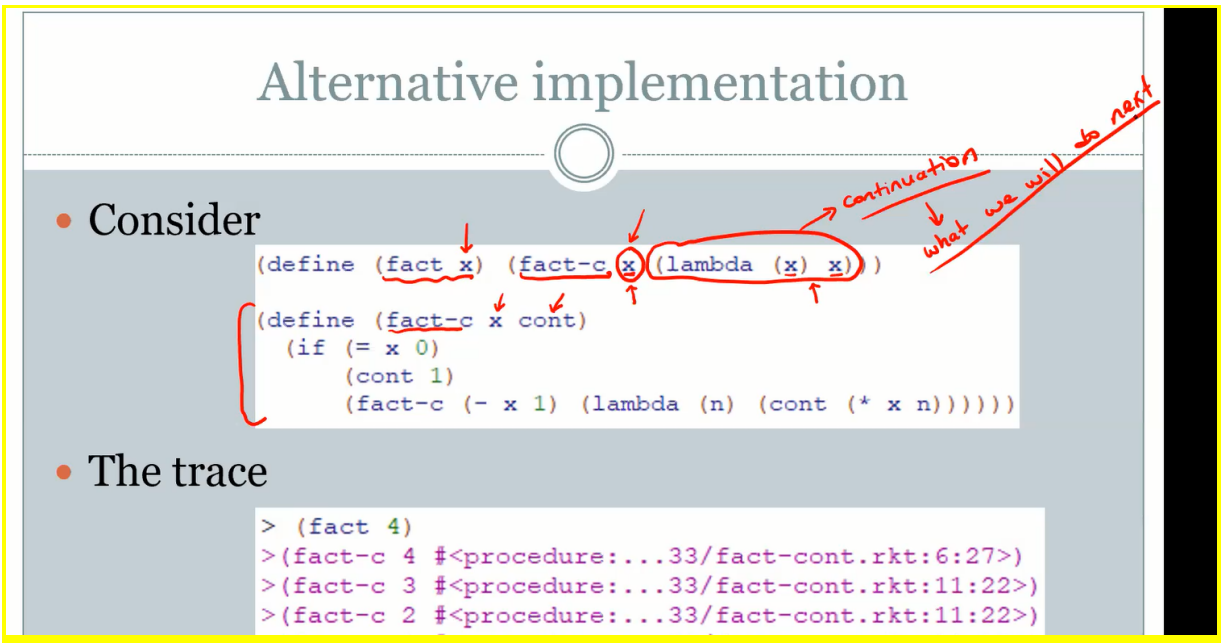
16:03

- 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 i=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
>
```

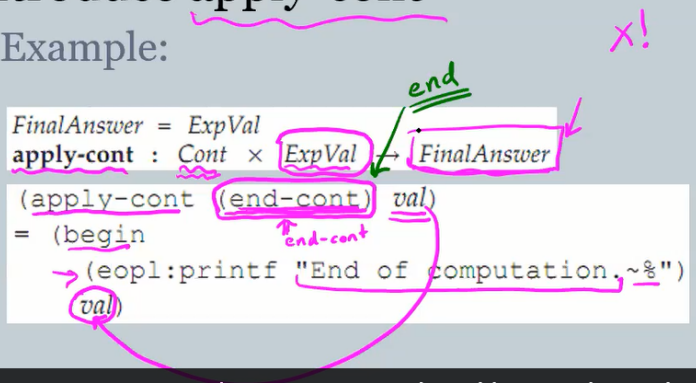


AAAAAAAAAA

WRITE A TAIL RECURSIVE VERISON OF A OVER B EXERCISEEEE

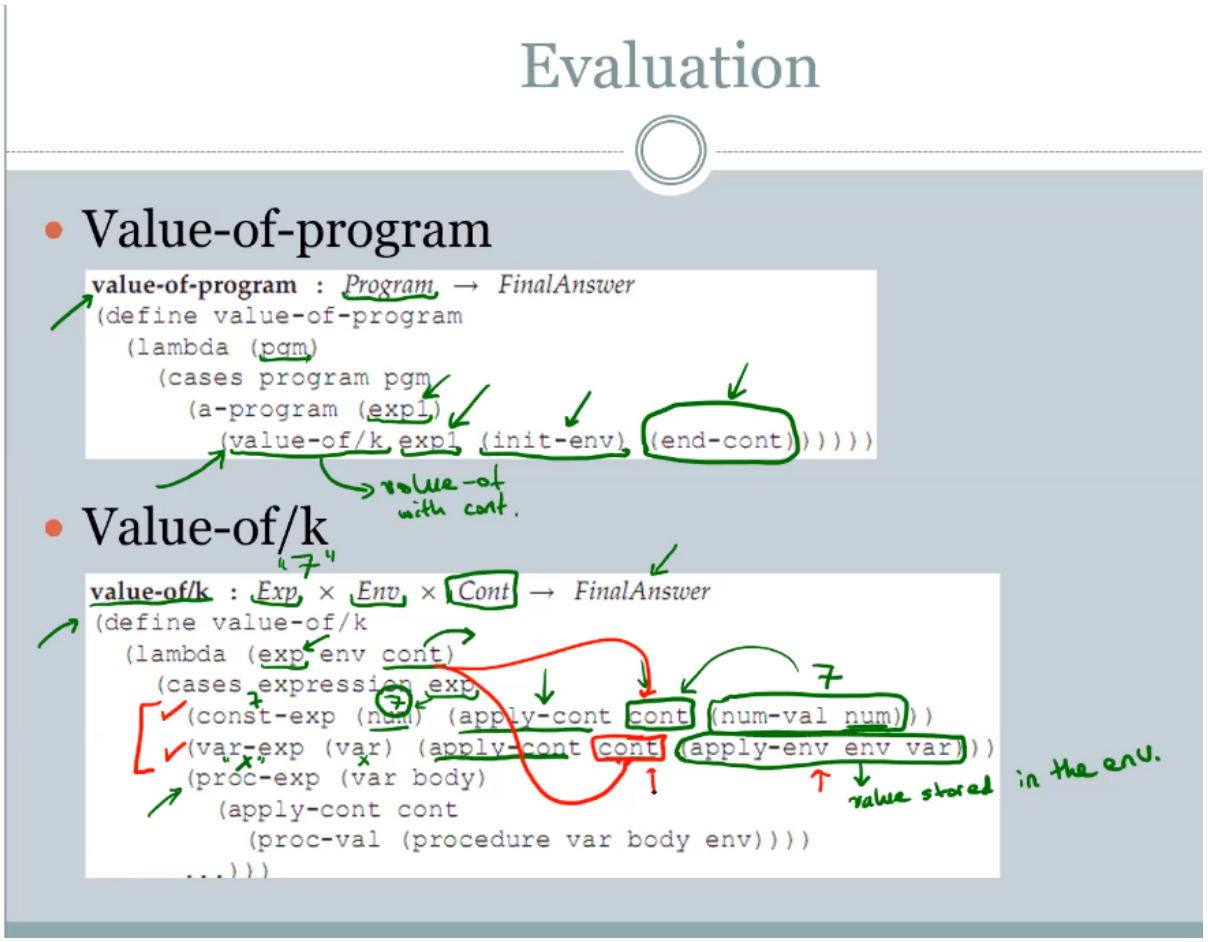
A CPS Interpreter

- 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:



AAAAA

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

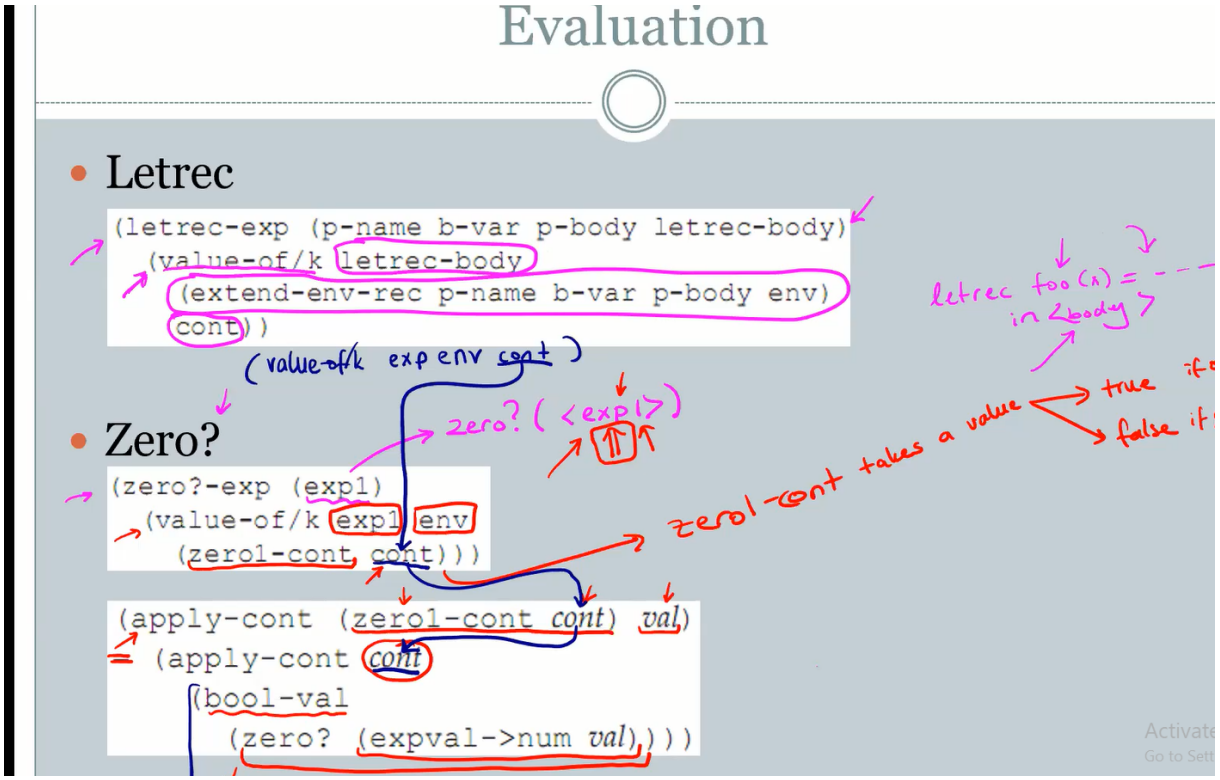


FOR THE ZERO?:

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



- 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 remem

:

Let exp cont:

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

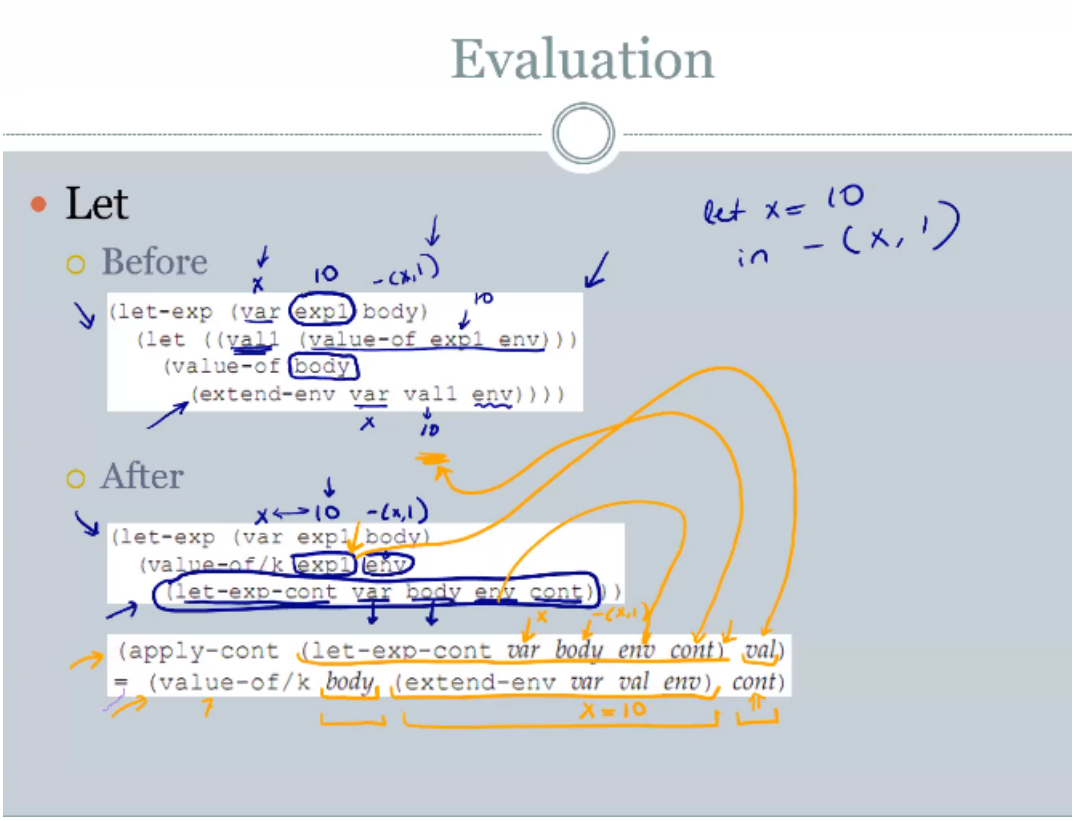
41:03

- 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

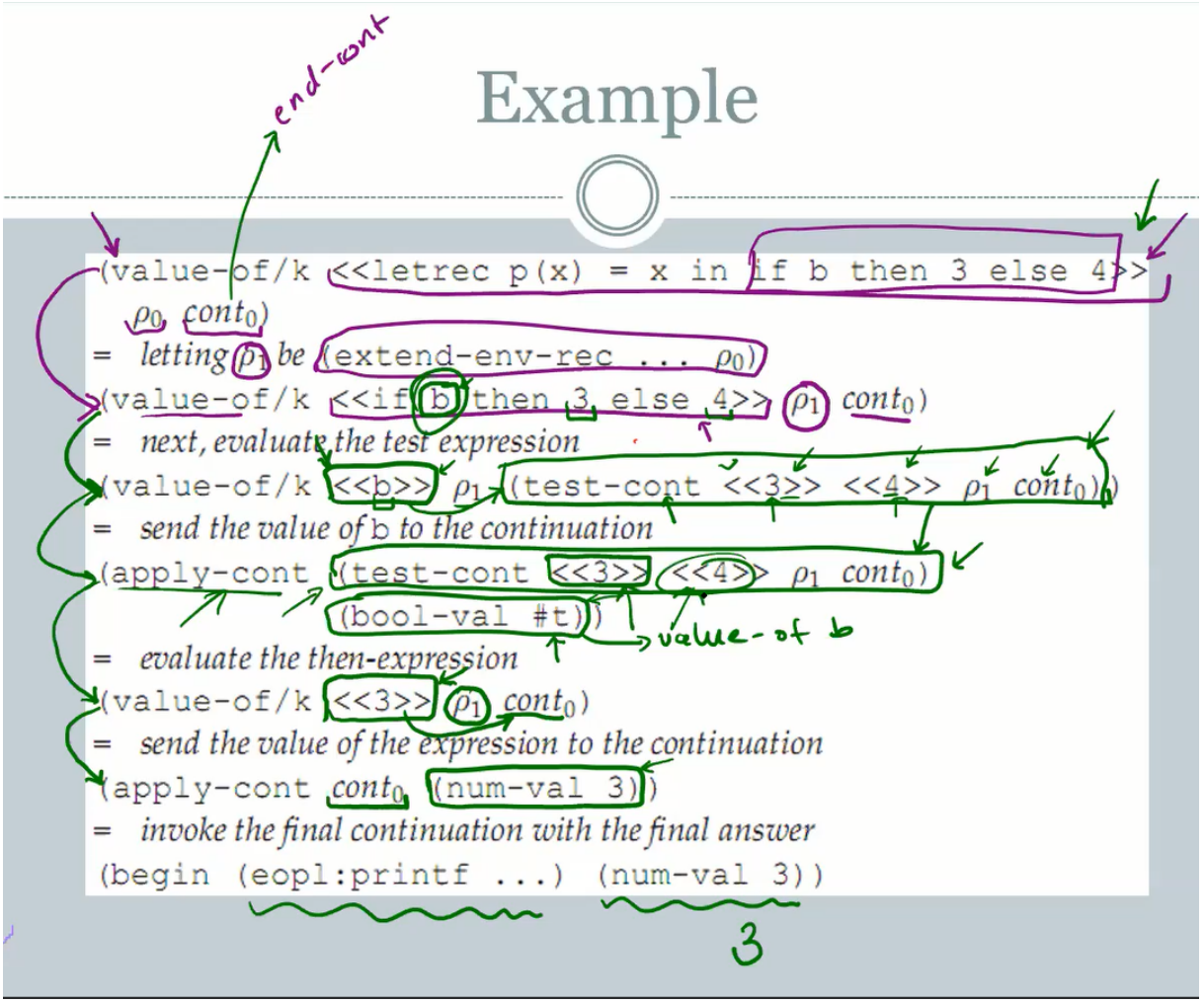
- BURCU YILDIZ: Here, right.

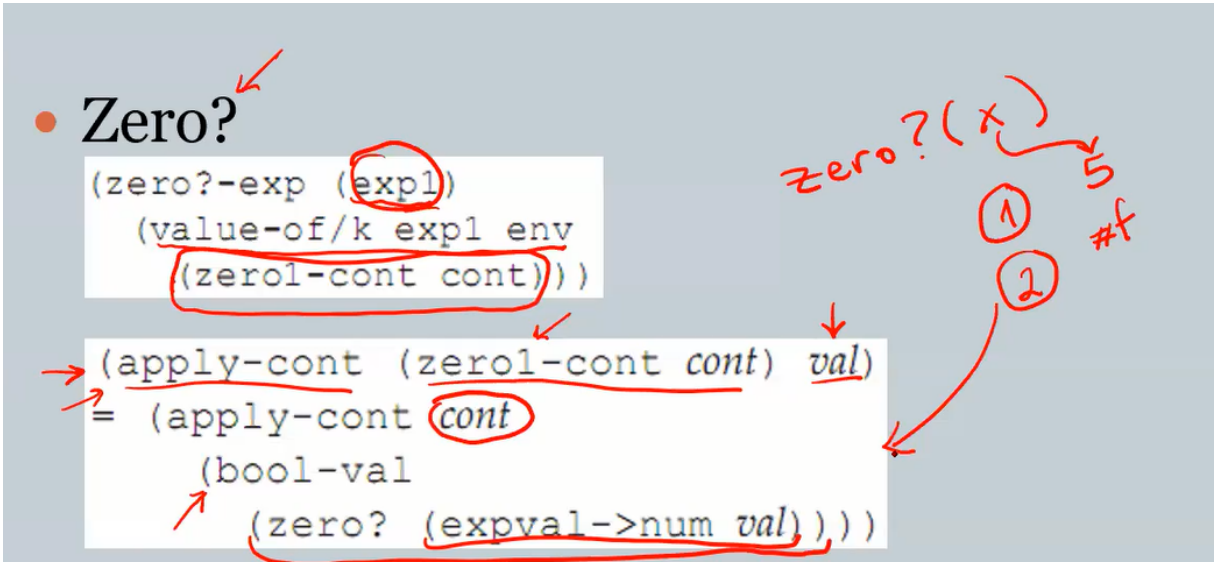
At the end it sendws it to cont (the next continuation):



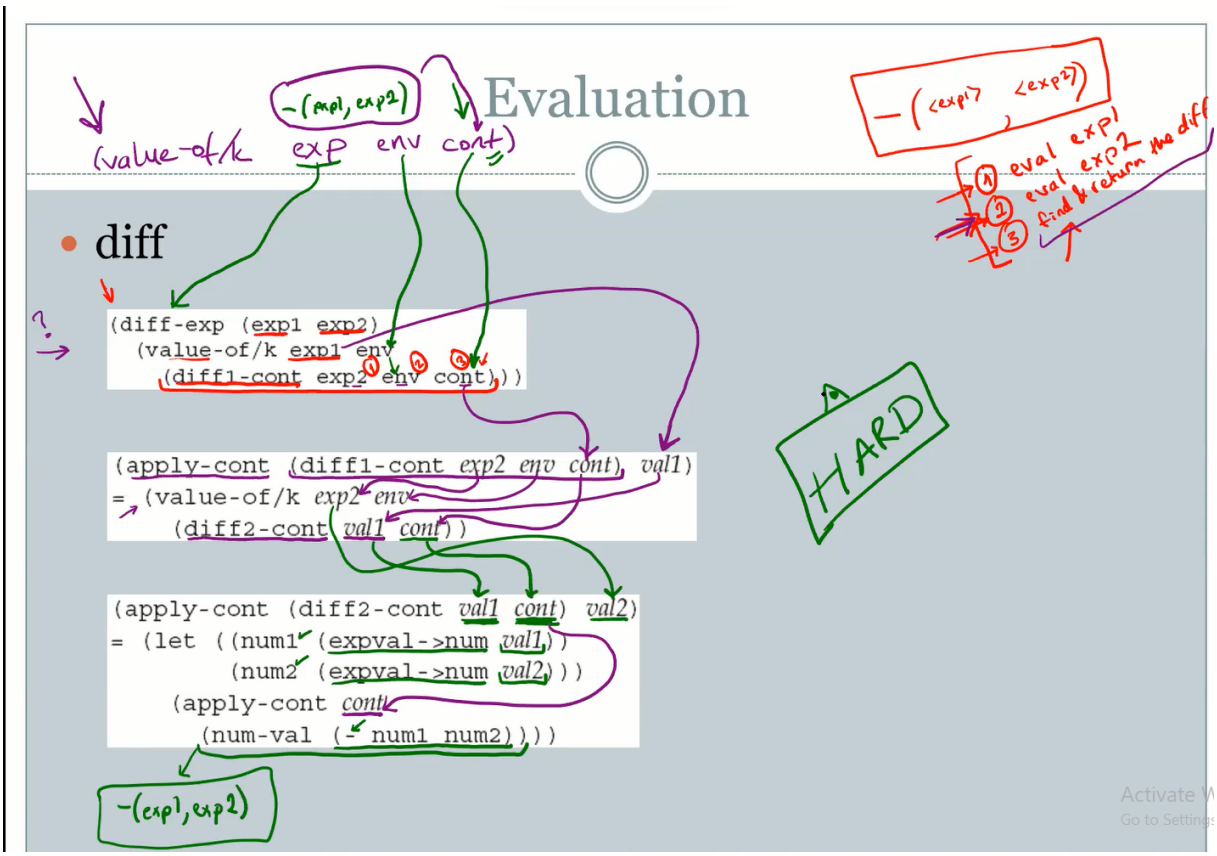
For if the main idea is it looks at the condition and THEN applies the continuation which is either the computation of exp2 or exp3 depending on the exp1 and whatever result comes out is pased to the cont being passed inside the below funciton which is the first continuation

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





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

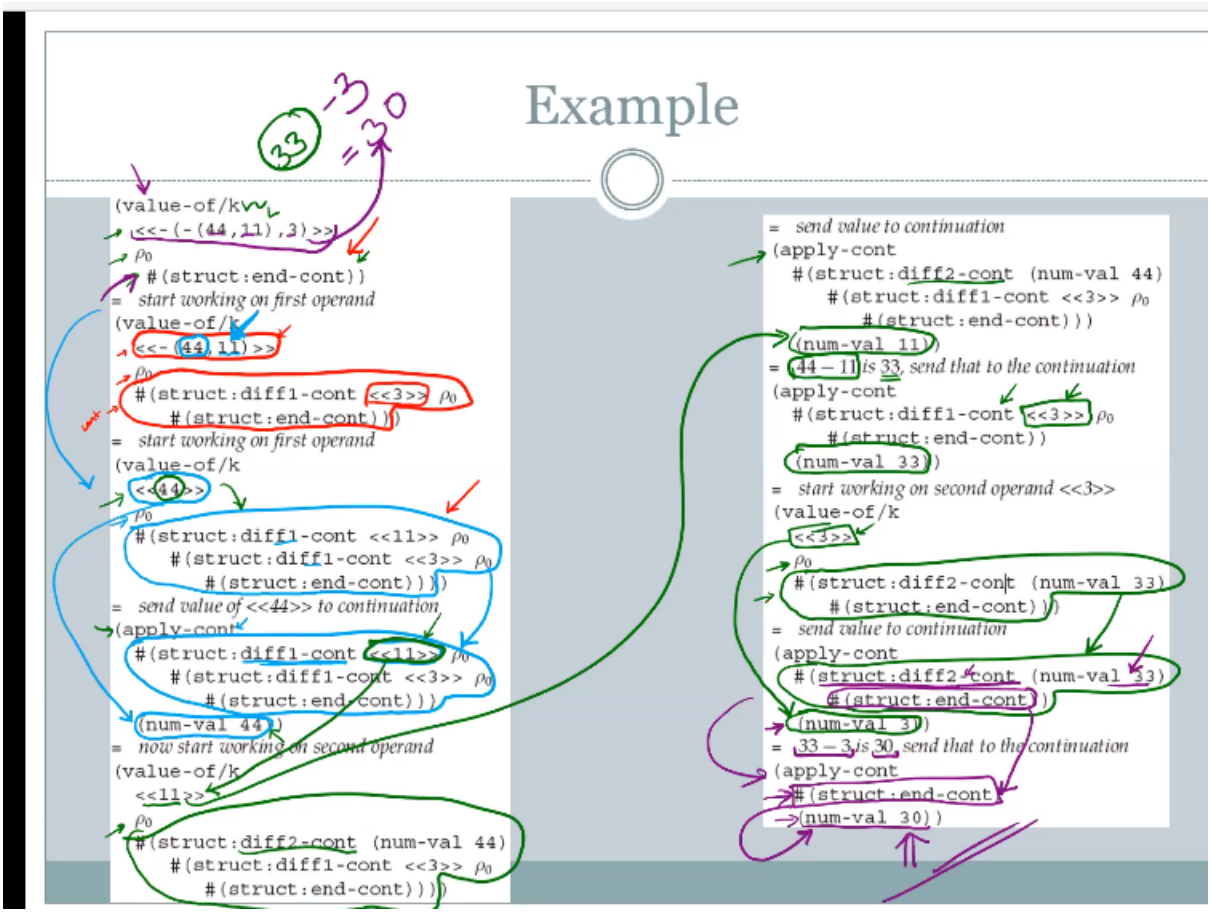


AAA

:

HERE BELOW, THE BELOW SECTION WITH THE STRUCTS REPRESENTS WHAT TO DO NEXT IN ORDE SO IN FIRTS BLUE CASE OF IT YOU EVALUATE44 THENT HE NEXT THUING TO DO IS TO EVALUATE 11 AND

THEN 3



Diff1continuation =evaluate the second expression

Diff2continuation = actually perform the substraction operation