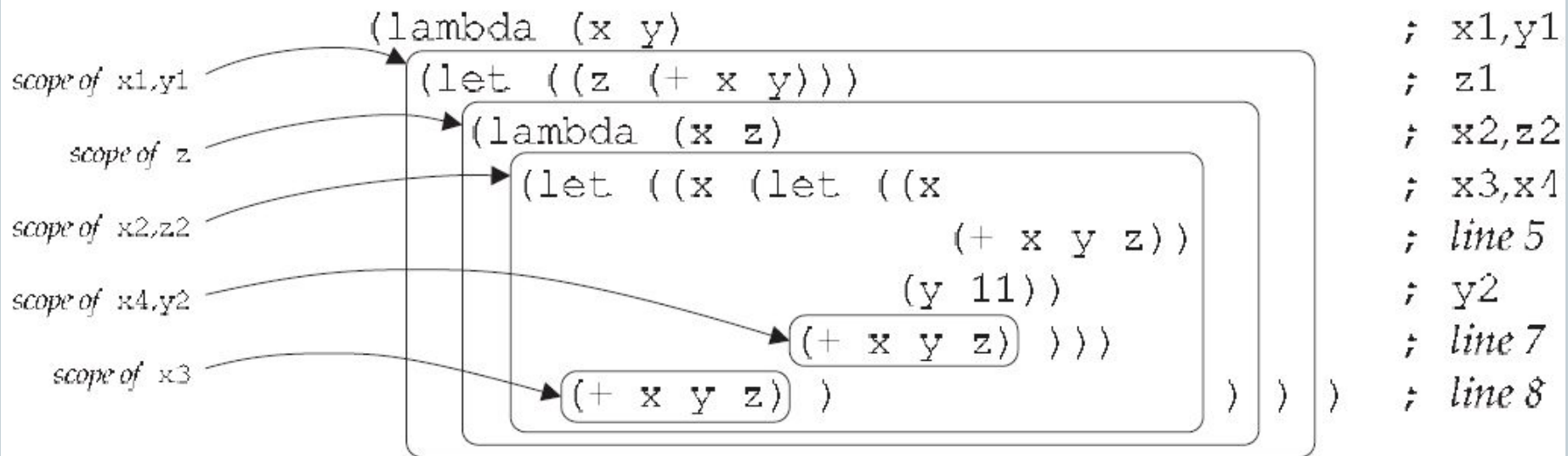


Scoping, Binding Lexical Addressing Review



T. METIN SEZGIN

Lexical depth



Nugget



**Arguments to procedures
always found at the expected
places**

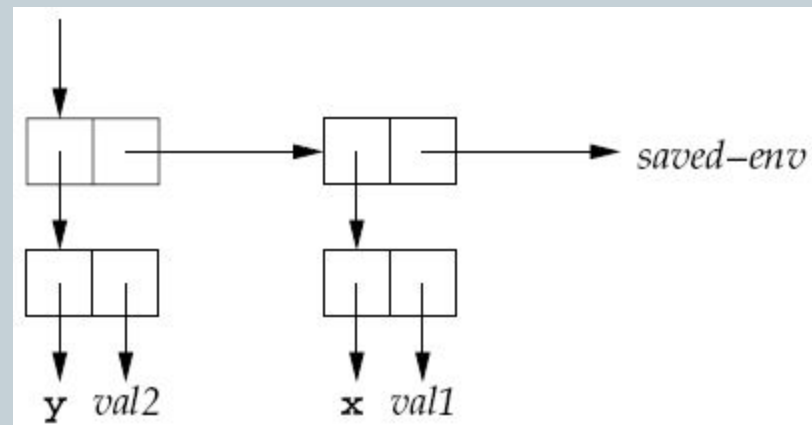
Evaluating expressions

- Consider the following execution trace:

```
let x = exp1  
in let y = exp2  
   in -(x,y)
```



```
(value-of  
  <<let x = exp1  
    in let y = exp2  
      in -(x,y)>>  
  ρ)  
=  
(value-of  
  <<let y = exp2  
    in -(x,y)>>  
  [x=val1] ρ)  
=  
(value-of  
  <<-(x,y)>>  
  [y=val2] [x=val1] ρ)
```



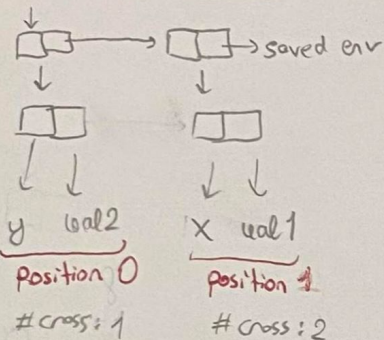
- let
 - letrec
 - proc

} defines declarations in our language.

let $x = \text{exp1}$
 in let $y = \text{exp2}$
 in $-(x, y)$

scope of x

scope of y



num of border crossing

lexical depth of the proc

represents the location in the env
 (0, 1, ...) starts from 0

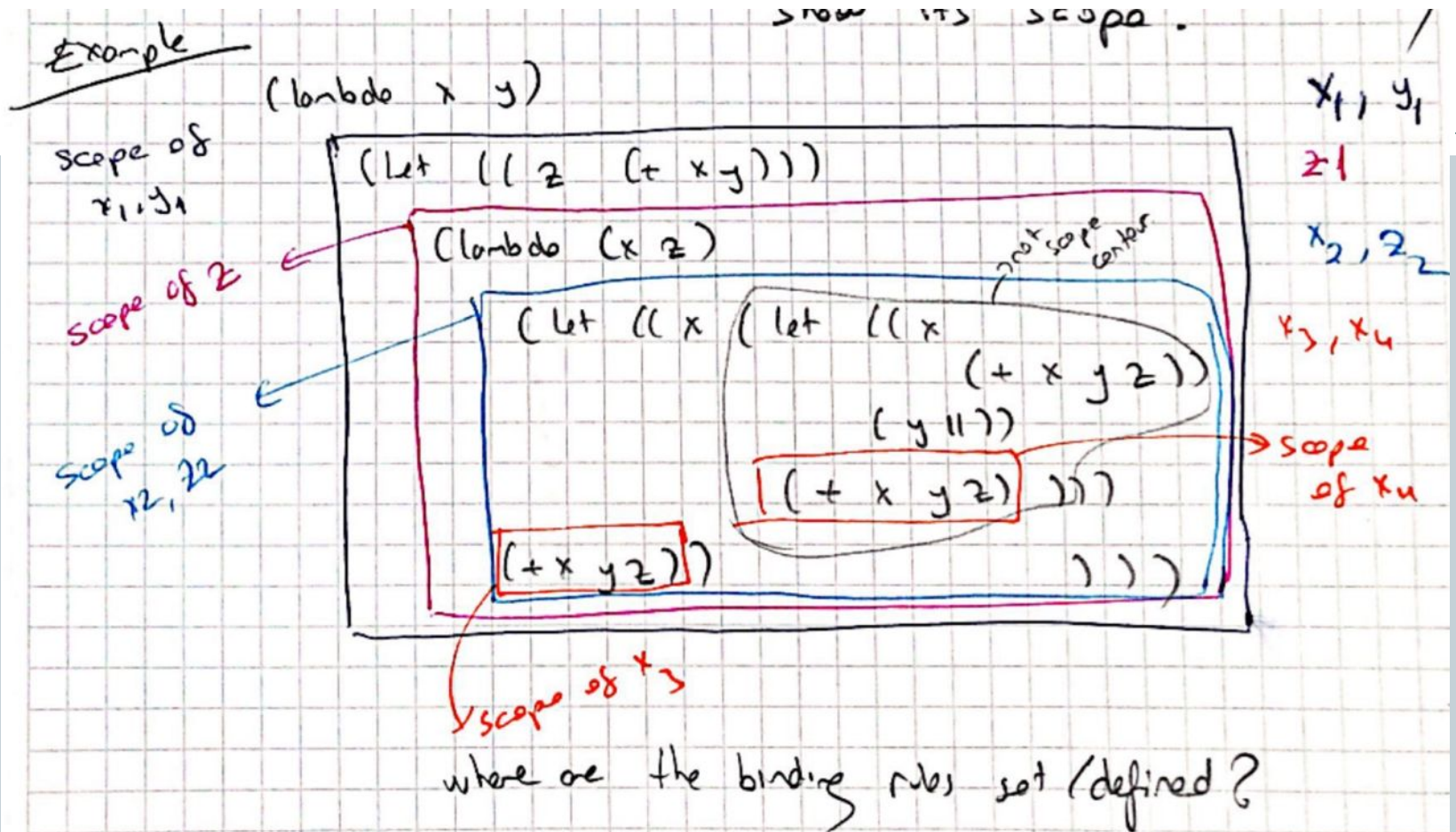
In the application part;

Ex: let $a = 5$
 in $\text{proc}(x) \text{ } -(x, a)$ } f

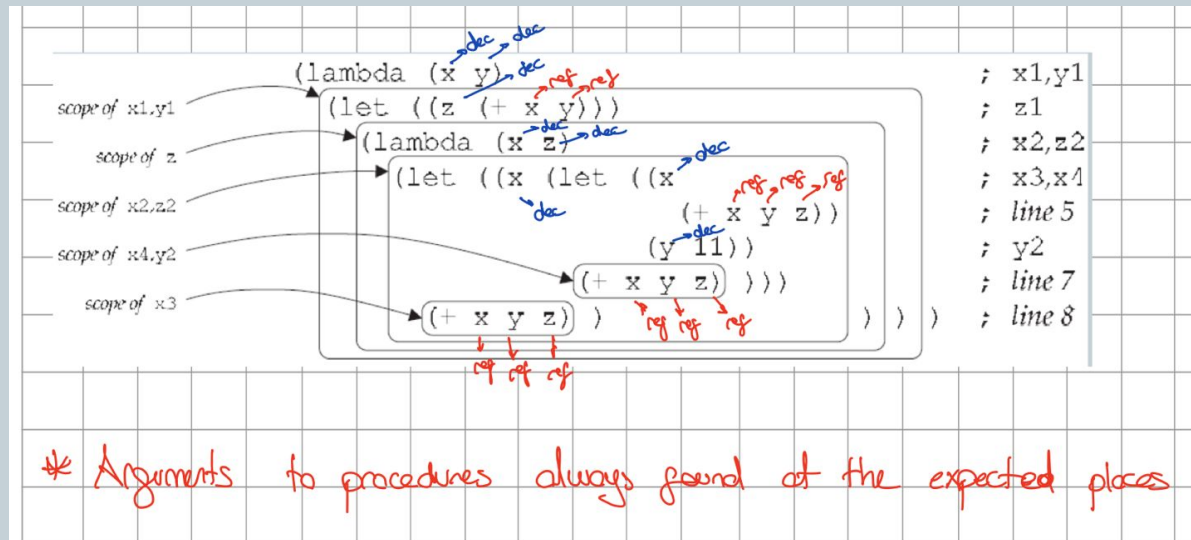
What is order when $(f \ 7)$ applied?

..... $[x = \lceil 7 \rceil] [a = \lceil 5 \rceil] p$
 ↓ ↓
 cross 1 cross 2
 position 0 position 1

Esra Duran



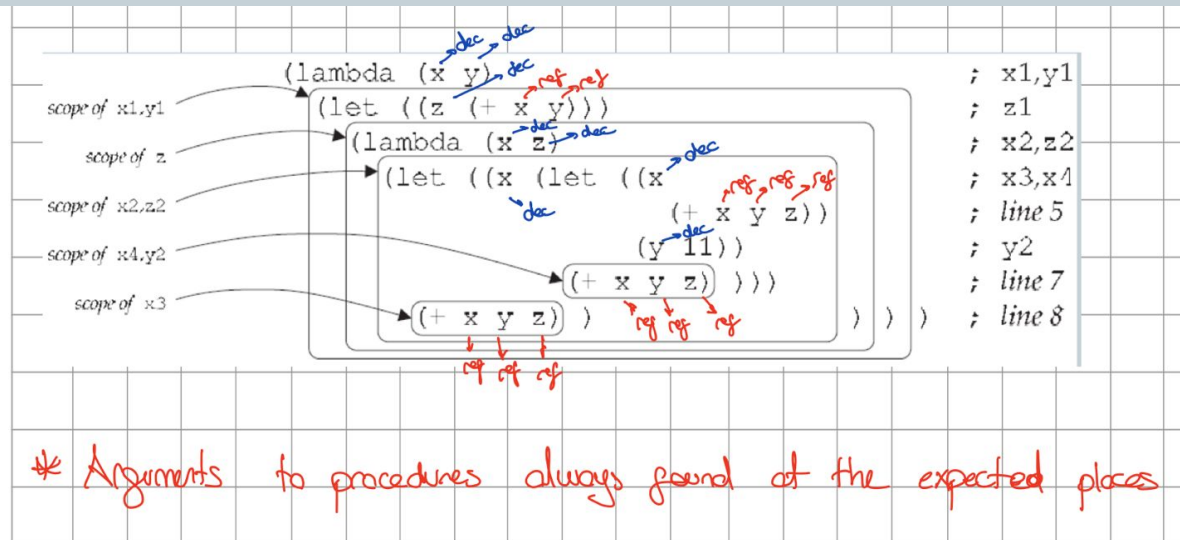
Deniz Yilmaz, Onur Mavitas
Cem Akarsubasi, Ozlem Olcer



Bora Karagul



This will be →
in the exam



Bora Karagul

* evaluating expressions

let x = exp₁
in let y = exp₂
in (x, y)

→ how does the environment look like when we are here?

- latest expansion $\Rightarrow y = \text{exp}_2$
- oldest expansion $\Rightarrow x = \text{exp}_1$

Dilara Yavuz

* evaluating expressions

```
let x = exp1
in let y = exp2
   in (x, y)
```

→ how does the environment look like when we are here?

- latest expansion $\Rightarrow y = \text{exp}_2$
- oldest expansion $\Rightarrow x = \text{exp}_1$

(sinavda bundan gılcık!!!)

Dilara Yavuz

⊛ evaluating expressions

```
let x = exp1
in let y = exp2
in (x, y)
```

→ how does the environment look like when we are here?

- latest expansion $\Rightarrow y = \text{exp}_2$
- oldest expansion $\Rightarrow x = \text{exp}_1$

(sinavda bundan gıcık!!)

* latest thing added is at the position 0 in env

* lexical depth:

- first border cross $\Rightarrow \text{depth} = 0$
- variable's lexical depth = the latest border cross to find that variable declaration
- positions in the environment is related to lexical depth.

(telefondaki resimleri gıcık!!)

Dilara Yavuz

Nameless Let Translation & Interpretation



T. METIN SEZGIN

Nuggets of the lecture



- Arguments to procedures always found at the expected places
- We don't need names
- We can create a new “nameless” language
- We can translate named language to the nameless one

Nugget



We don't need names

We don't need names



- We can create a new “nameless” language

```
(lambda (x)
  ((lambda (a)
    (x a))
   x))
```



```
(nameless-lambda
  ((nameless-lambda
    (#1 #0))
   #0))
```


Implementing lexical addressing



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

```
let x = 37
in proc (y)
  let z = -(y,x)
  in -(x,y)
```



```
#(struct:a-program
  #(struct:nameless-let-exp
    #(struct:const-exp 37)
    #(struct:nameless-proc-exp
      #(struct:nameless-let-exp
        #(struct:diff-exp
          #(struct:nameless-var-exp 0)
          #(struct:nameless-var-exp 1))
        #(struct:diff-exp
          #(struct:nameless-var-exp 2)
          #(struct:nameless-var-exp 1))))))
```

Nugget



**We can create a new
“nameless” language**

The translator: the target language



Expression ::= %lexref *number*

`nameless-var-exp (num)`

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

`nameless-let-exp (exp1 body)`

Expression ::= %lexproc *Expression*

`nameless-proc-exp (body)`

Nugget



**We can translate the named
language to the nameless one**

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



Static Environment

$\text{Senv} = \text{Listof}(\text{Sym})$

$\text{Lexaddr} = N$

$\text{empty-senv} : () \rightarrow \text{Senv}$

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

$\text{extend-senv} : \text{Var} \times \text{Senv} \rightarrow \text{Senv}$

```
(define extend-senv
  (lambda (var senv)
    (cons var senv)))
```

$\text{apply-senv} : \text{Senv} \times \text{Var} \rightarrow \text{Lexaddr}$

```
(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  $\rightarrow$  Nameless-program
(define translation-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (a-program
          (translation-of exp1 (init-senv)))))))

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

Translator 2



translation-of : $Exp \times Senv \rightarrow Nameless-exp$

```
(define translation-of
  (lambda (exp senv)
    (cases expression exp
      (const-exp (num) (const-exp num))
      (diff-exp (exp1 exp2)
        (diff-exp
          (translation-of exp1 senv)
          (translation-of exp2 senv)))
      (zero?-exp (exp1)
        (zero?-exp
          (translation-of exp1 senv)))
      (if-exp (exp1 exp2 exp3)
        (if-exp
          (translation-of exp1 senv)
          (translation-of exp2 senv)
          (translation-of exp3 senv)))
```

```
      (var-exp (var)
        (nameless-var-exp
          (apply-senv senv var)))
      (let-exp (var exp1 body)
        (nameless-let-exp
          (translation-of exp1 senv)
          (translation-of body
            (extend-senv var senv))))
      (proc-exp (var body)
        (nameless-proc-exp
          (translation-of body
            (extend-senv var senv))))
      (call-exp (rator rand)
        (call-exp
          (translation-of rator senv)
          (translation-of rand senv)))
      (else
        (report-invalid-source-expression exp))))
```


Interpretation



Nameless interpreter



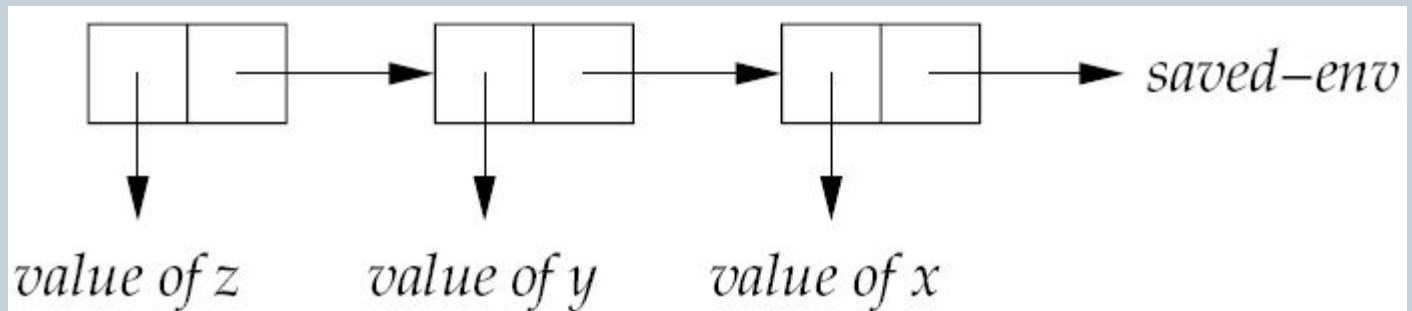
```
run : String → ExpVal
(define run
  (lambda (string)
    (value-of-program
     (translation-of-program
      (scan&parse string))))))
```

New environment interface



nameless-environment

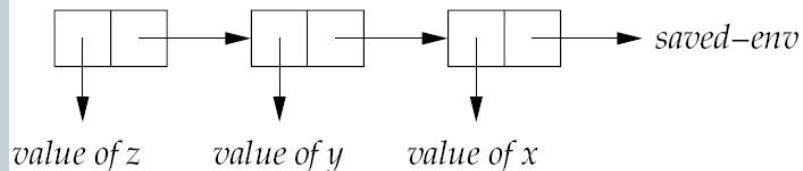
nameless-environment? : $SchemeVal \rightarrow Bool$
empty-nameless-env : $() \rightarrow Nameless-env$
extend-nameless-env : $Expval \times Nameless-env \rightarrow Nameless-env$
apply-nameless-env : $Nameless-env \times Lexaddr \rightarrow DenVal$



New environment interface



nameless-environment? : $SchemeVal \rightarrow Bool$
empty-nameless-env : $() \rightarrow Nameless-env$
extend-nameless-env : $Expval \times Nameless-env \rightarrow Nameless-env$
apply-nameless-env : $Nameless-env \times Lexaddr \rightarrow DenVal$



```
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)))
```

Procedure specification and implementation



```
(apply-procedure (procedure body  $\rho$ ) val)  
= (value-of body (extend-nameless-env val  $\rho$ ))
```

```
procedure : Nameless-exp  $\times$  Nameless-env  $\rightarrow$  Proc  
(define-datatype proc proc?  
  (procedure  
    (body expression?)  
    (saved-nameless-env nameless-environment?)))
```

```
apply-procedure : Proc  $\times$  ExpVal  $\rightarrow$  ExpVal  
(define apply-procedure  
  (lambda (proc1 val)  
    (cases proc proc1  
      (procedure (body saved-nameless-env)  
        (value-of body  
          (extend-nameless-env val saved-nameless-env))))))
```

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))))))
```

Quiz #9



1. Find the value
2. Draw the contours for the entire expression
3. Draw the environment diagram at the time of evaluation for the expression in the box

```
let x = exp1  
in let y = exp2  
   in - (x, y)
```