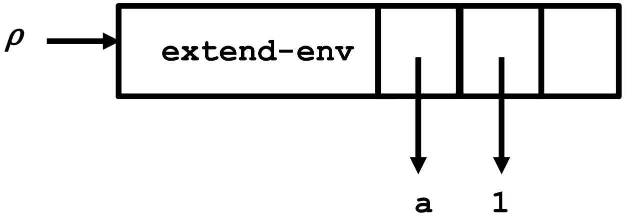
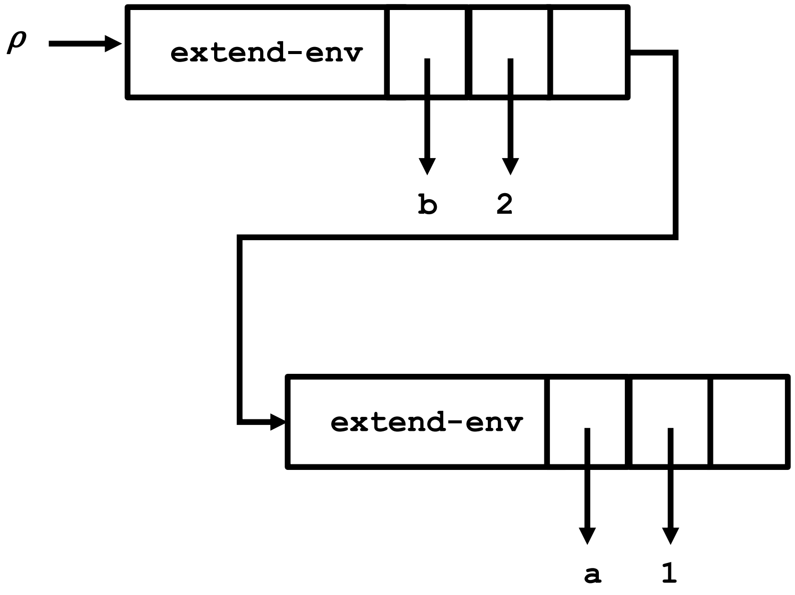
Letrec Review

T. METIN SEZGIN

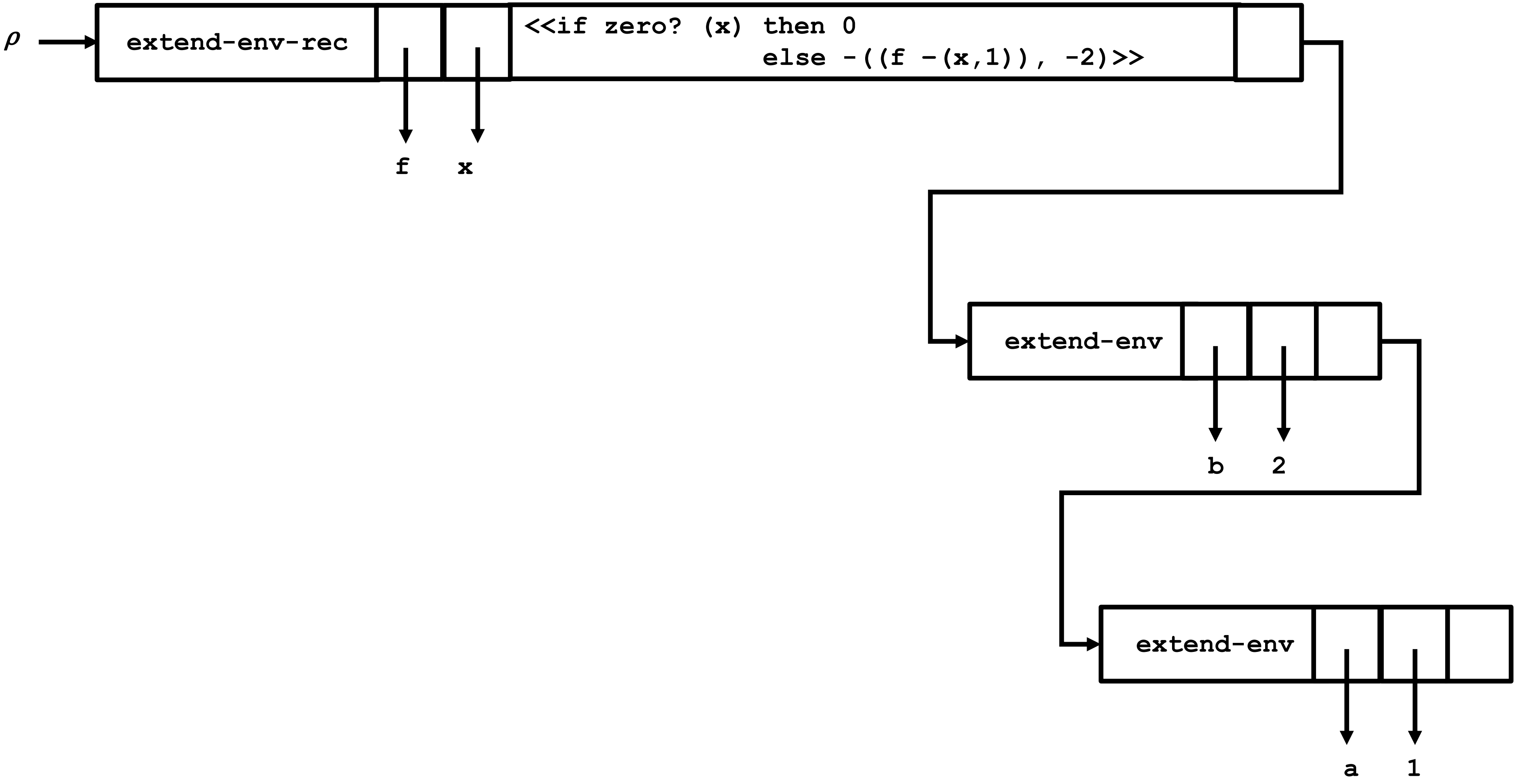
```
let a=1 in
  let b=2 in
    letrec f(x) = if zero?(x) then 0 else -((f -(x,1)),-2) in (f 2)
```



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



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



ρ

extend-env-rec

<<if zero? (x) then 0
else -((f -(x,1)), -2)>>

f x

extend-env

b 2

extend-env

a 1

Scoping, Binding
Lexical Addressing

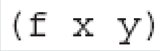
T. METIN SEZGIN

Nuggets of the lecture

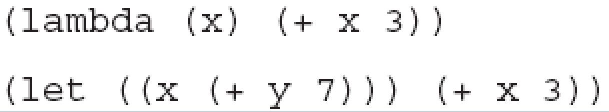
- Scoping controls how values are bound to variables
- Arguments to procedures always found at the expected places
- We don’t need names
- We can create a new “nameless” language
- We can translate named language to the nameless one

Denoted values

- Variables
  - References
  - Declarations



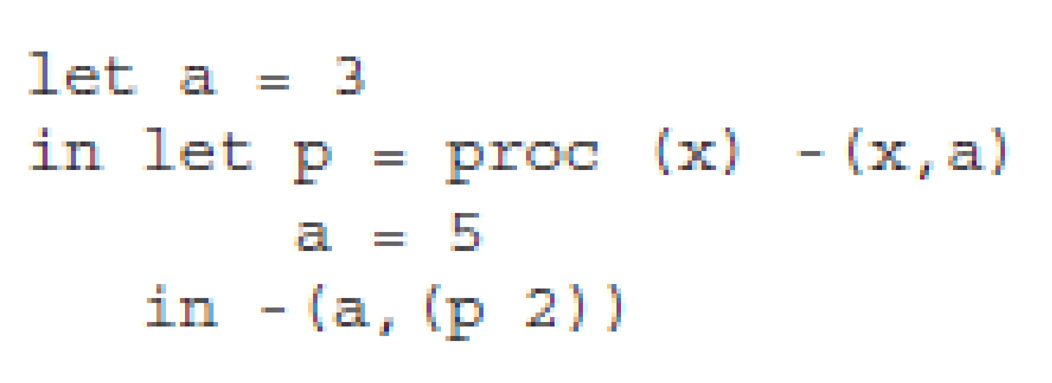
```
(f x y)
```



```
(lambda (x) (+ x 3))

(let ((x (+ y 7))) (+ x 3))
```

- Semantics
  - Binding
  - Scope

What is the value of this expression?

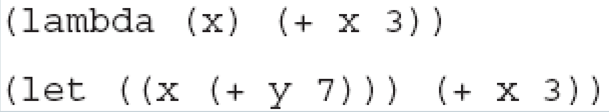


```
let a = 3
in let p = proc (x) -(x,a)
       a = 5
   in -(a,(p 2))
```

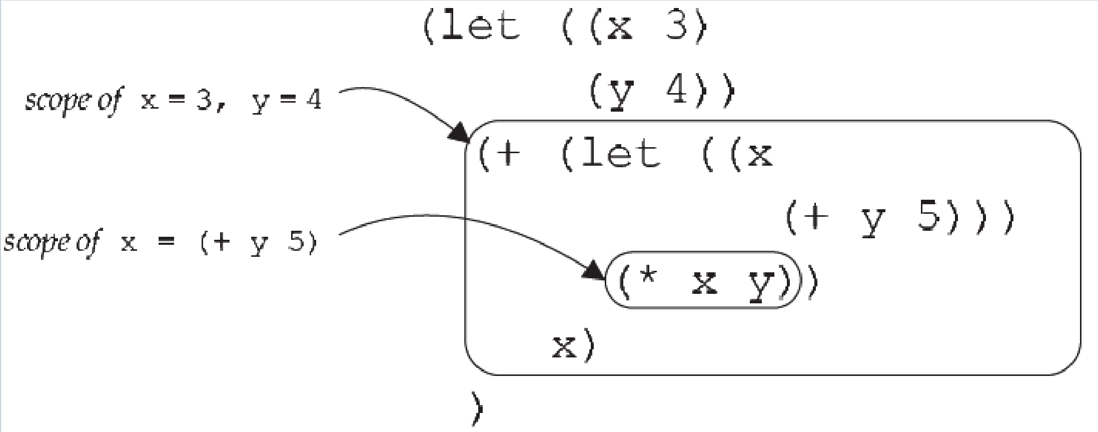
Denoted values

- Variables
  - References
  - Declarations
- Semantics
  - Binding
  - Scope

```
(f x y)
```



```
(lambda (x) (+ x 3))

(let ((x (+ y 7))) (+ x 3))
```

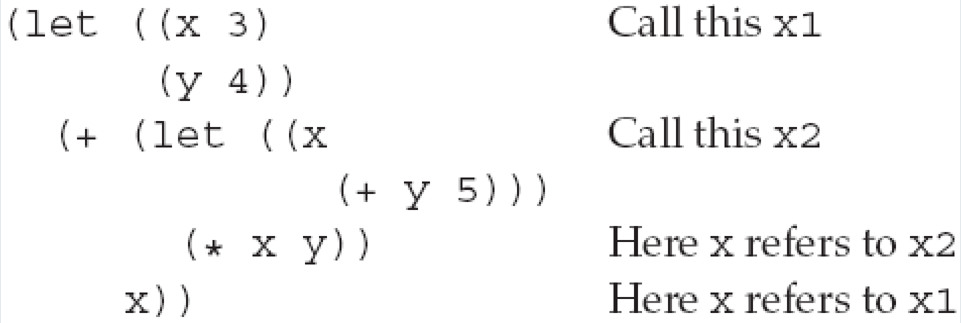


```
                (let ((x 3)
                      (y 4))
                  (+ (let ((x
                             (+ y 5)))
                       (* x y))
                     x)
                )
```

we need rules to define scoping

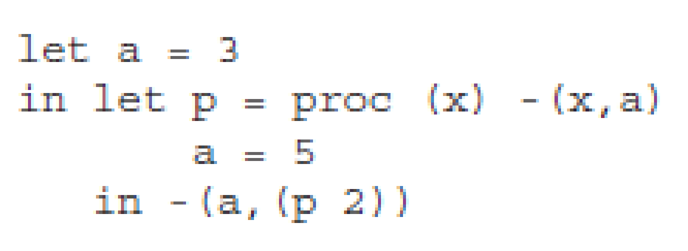
Scoping

- Static scoping
  - Declarations and references can be matched without code execution
  - Search “outward”



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

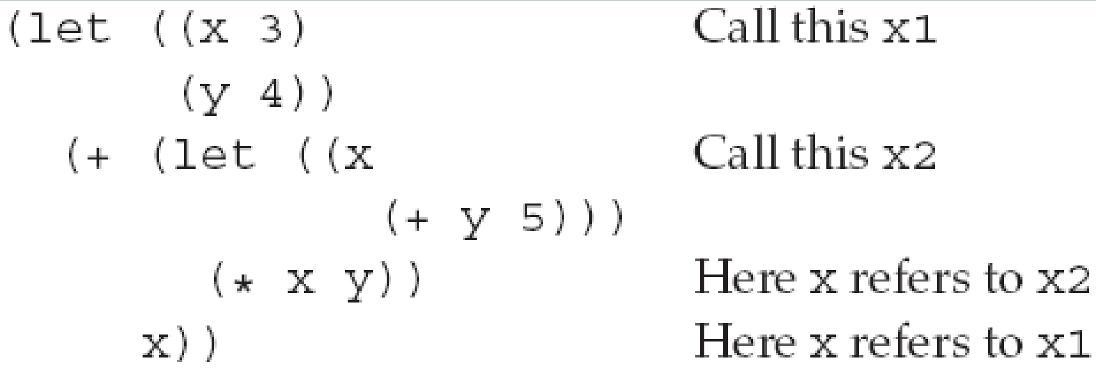
- Dynamic scoping
  - Declarations and references are matched during code execution
  - a in the proc bound to 5



```
let a = 3
in let p = proc (x) -(x,a)
       a = 5
   in -(a,(p 2))
```

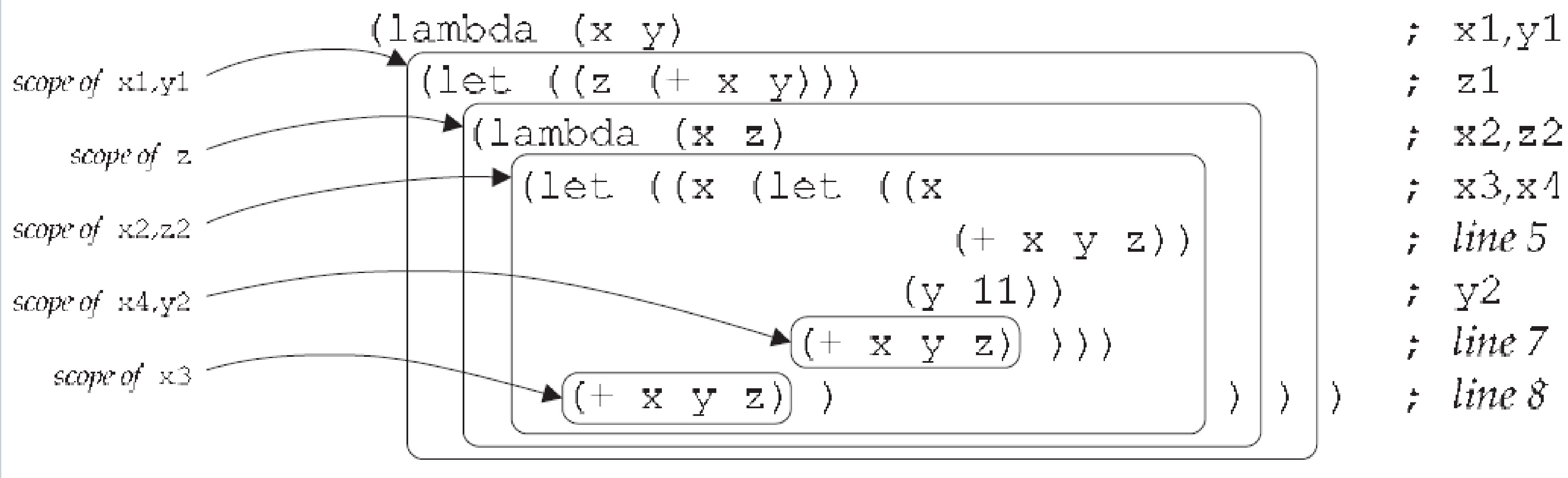
Concepts

- Shadowing
- Holes
- Extent
  - Duration of the binding
- Contour diagram
  - Helps resolving bindings
- Lexical depth



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

Another example



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

scope of x1,y1

scope of z

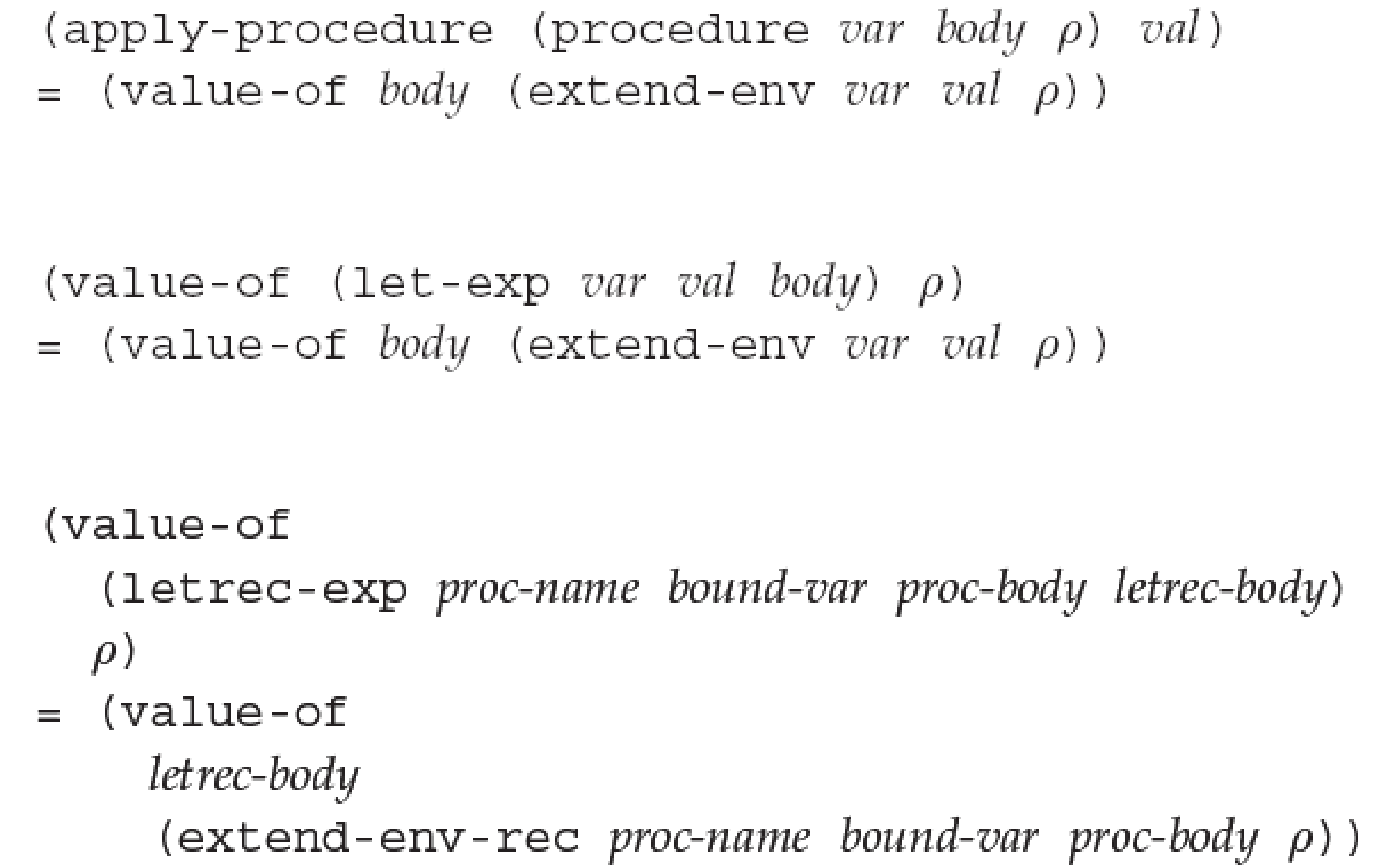
scope of x2,z2

scope of x4,y2

scope of x3

Where are the binding rules set/defined?

How are the binding rules defined?



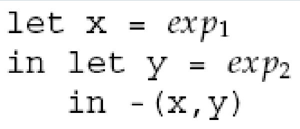
```
(apply-procedure (procedure var body ρ) val)
= (value-of body (extend-env var val ρ))



(value-of (let-exp var val body) ρ)
= (value-of body (extend-env var val ρ))



(value-of
  (letrec-exp proc-name bound-var proc-body letrec-body)
  ρ)
= (value-of
    letrec-body
    (extend-env-rec proc-name bound-var proc-body ρ))
```

Nugget

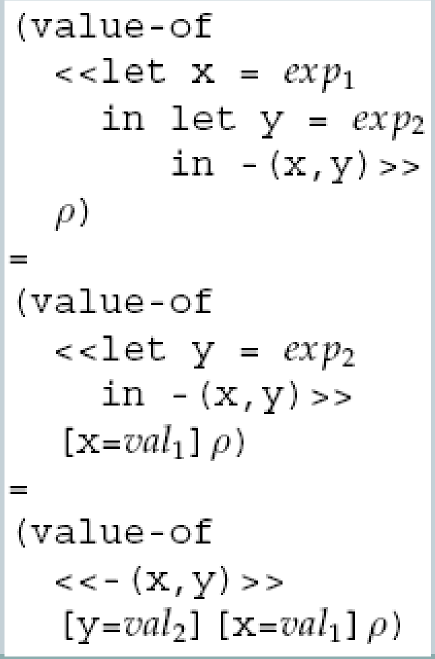
Arguments to procedures always found at the expected places

Evaluating expressions

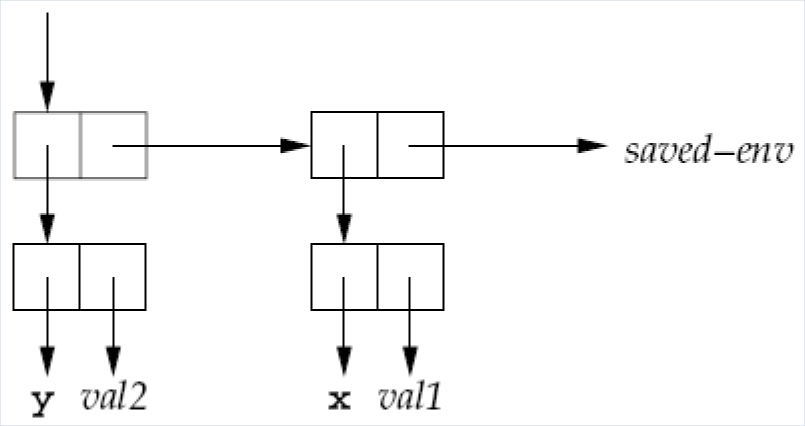
- Consider the following execution trace:



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

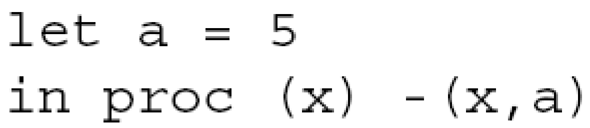


```
(value-of
  <<let x = exp1
     in let y = exp2
          in -(x,y)>>
  ρ)
=
(value-of
  <<let y = exp2
     in -(x,y)>>
   [x=val1]ρ)
=
(value-of
  <<-(x,y)>>
   [y=val2][x=val1]ρ)
```



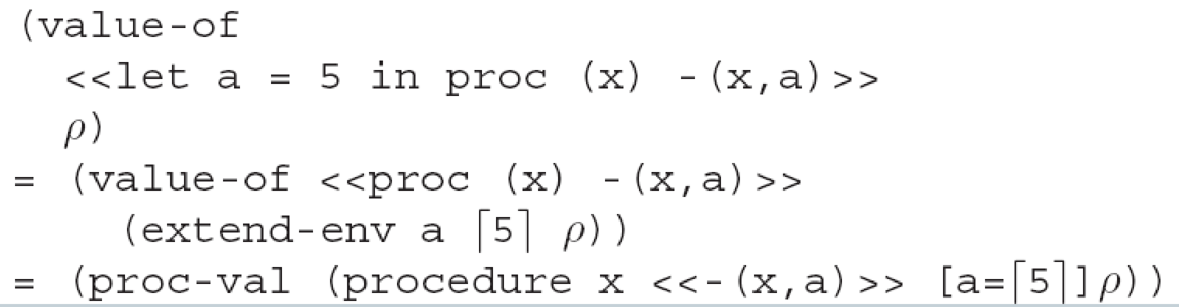
Consider another example

- The expression:



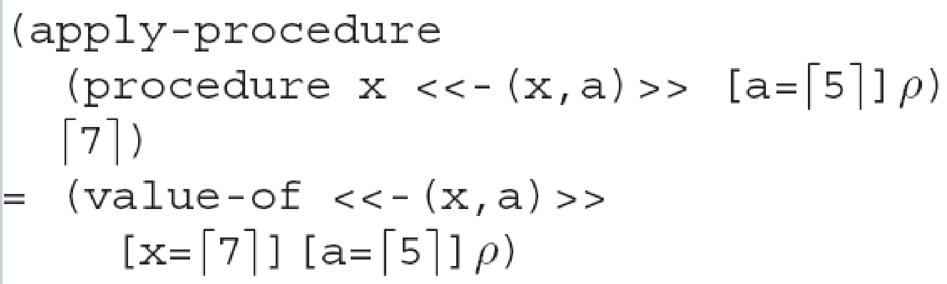
```
let a = 5
in proc (x) -(x,a)
```

- Its value:



```
(value-of
  <<let a = 5 in proc (x) -(x,a)>>
  ρ)
= (value-of <<proc (x) -(x,a)>>
    (extend-env a ⌈5⌉ ρ))
= (proc-val (procedure x <<-(x,a)>> [a=⌈5⌉]ρ))
```

- Application:



```
(apply-procedure
  (procedure x <<-(x,a)>> [a=⌈5⌉]ρ)
  ⌈7⌉)
= (value-of <<-(x,a)>>
    [x=⌈7⌉][a=⌈5⌉]ρ)
```

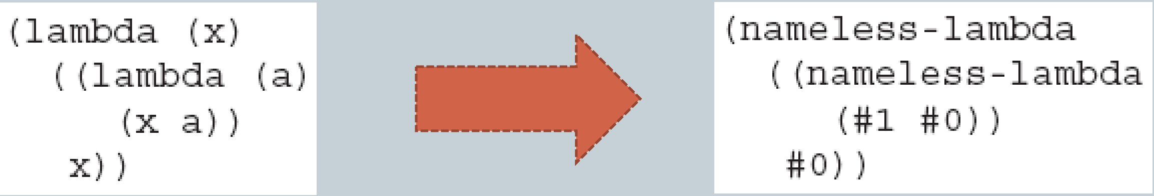
Things are found at the expected lexical depth!

Nugget

We don’t need names

We don’t need names

- We can create a new “nameless” language

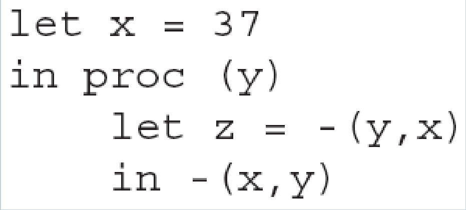


```
(lambda (x)
  ((lambda (a)
      (x a))
   x))
```

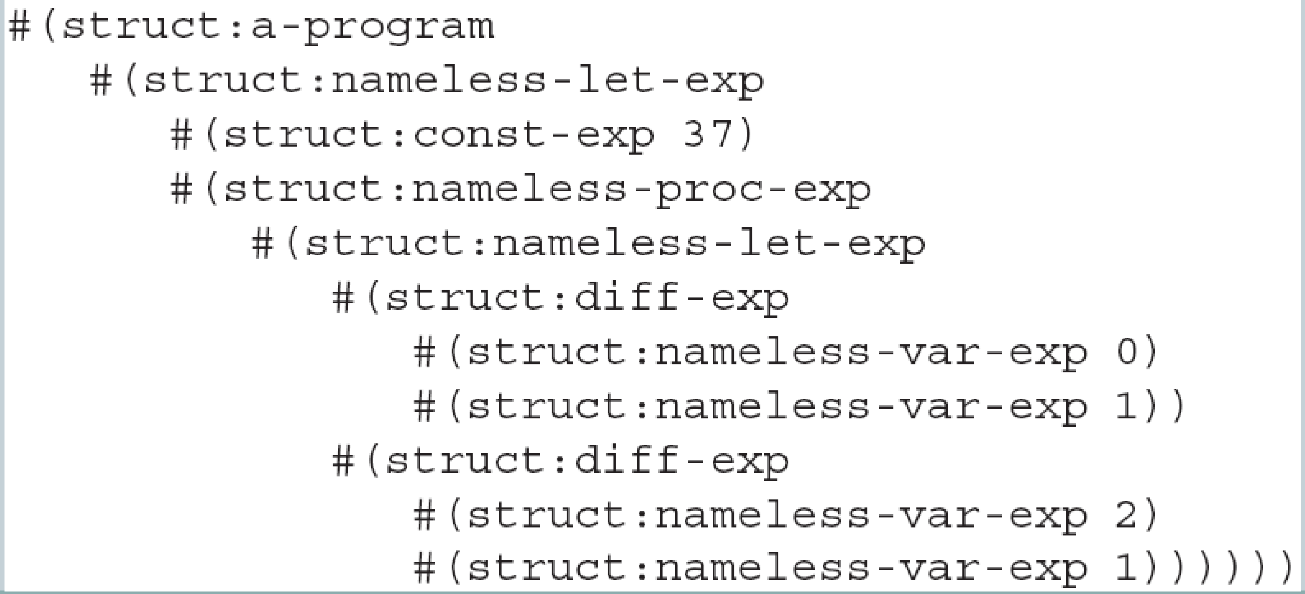
```
(nameless-lambda
  ((nameless-lambda
      (#1 #0))
   #0))
```

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)
```

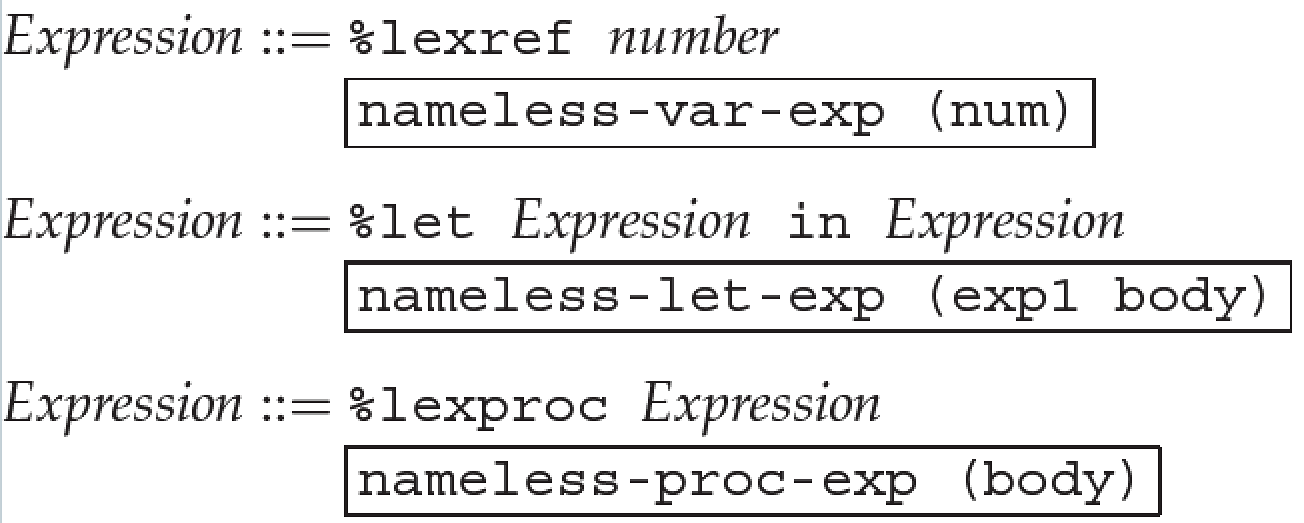


```
#(struct:a-program
   #(struct:nameless-let-exp
      #(struct:const-exp 37)
      #(struct:nameless-proc-exp
         #(struct:nameless-let-exp
            #(struct:diff-exp
               #(struct:nameless-var-exp 0)
               #(struct:nameless-var-exp 1))
            #(struct:diff-exp
  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

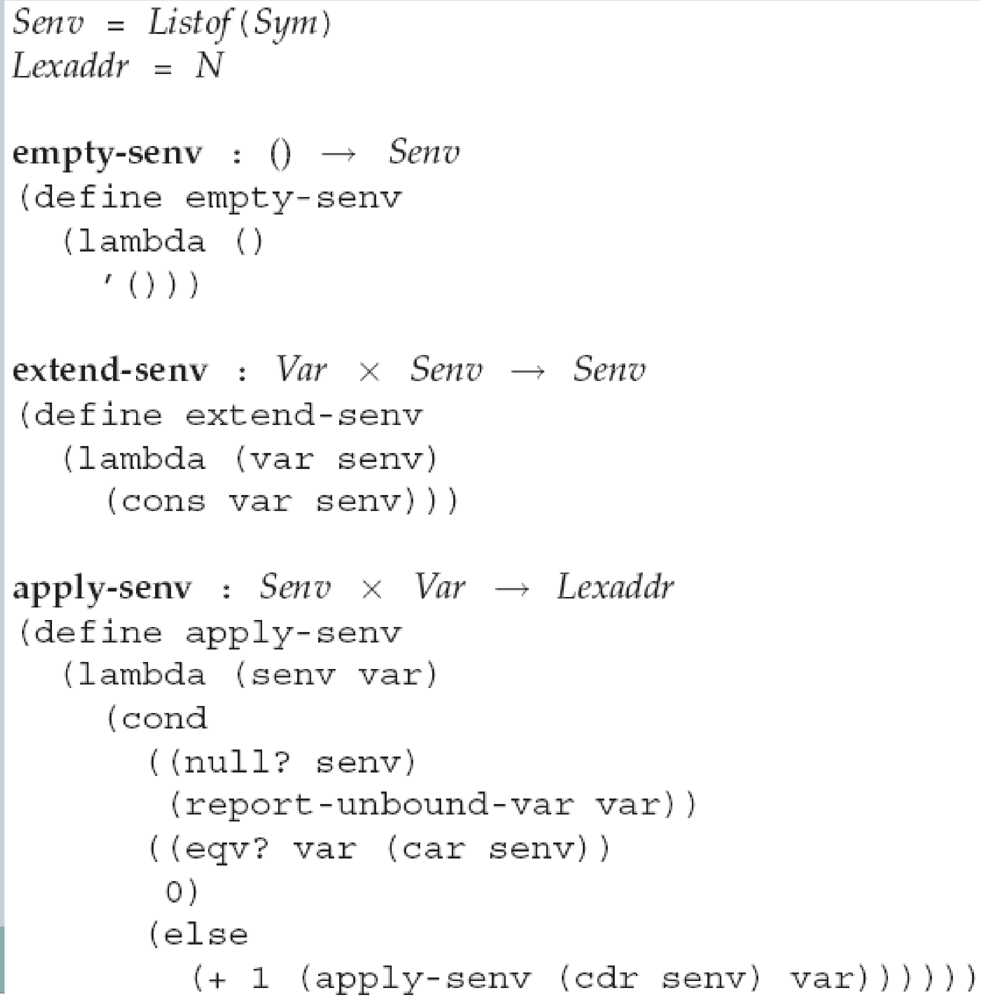
`nameless-proc-exp (body)`

Nugget

We can translate the named language to the nameless one

The translator: Exp x Senv → NamelessExp

Static Environment



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

empty-senv : $() \rightarrow Senv$

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

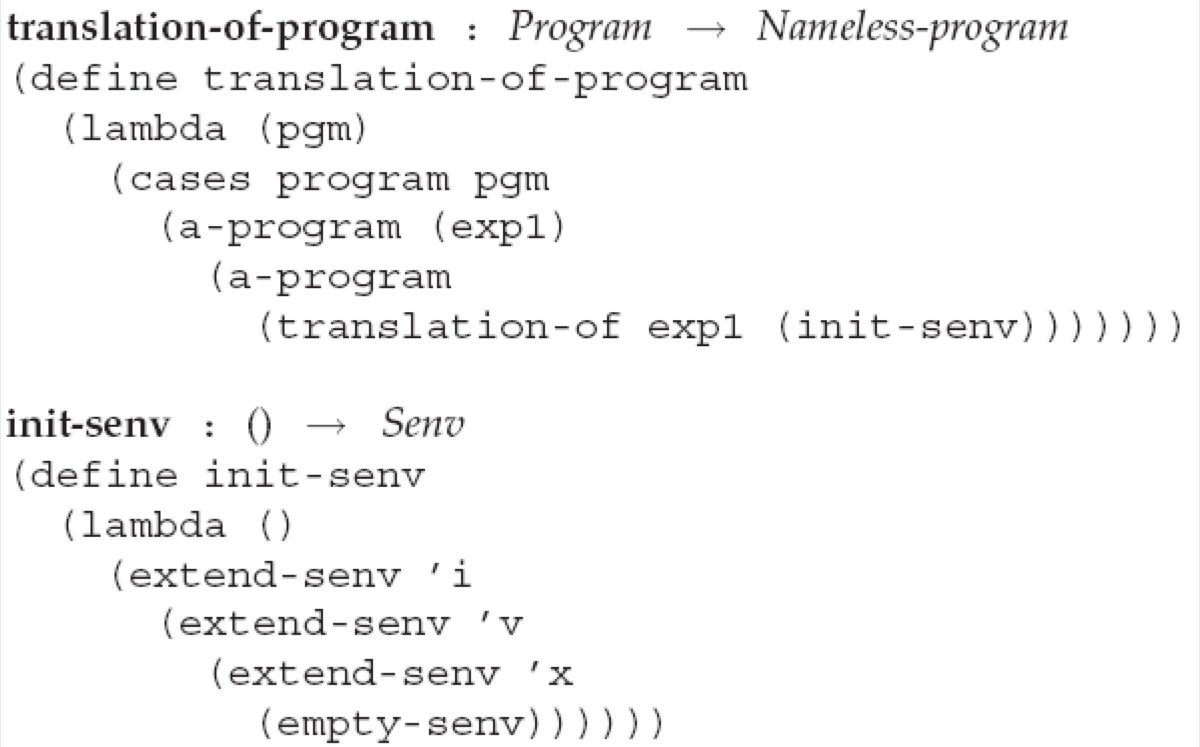
extend-senv : $Var \times Senv \rightarrow Senv$

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

apply-senv : $Senv \times Var \rightarrow Lexaddr$

```
(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



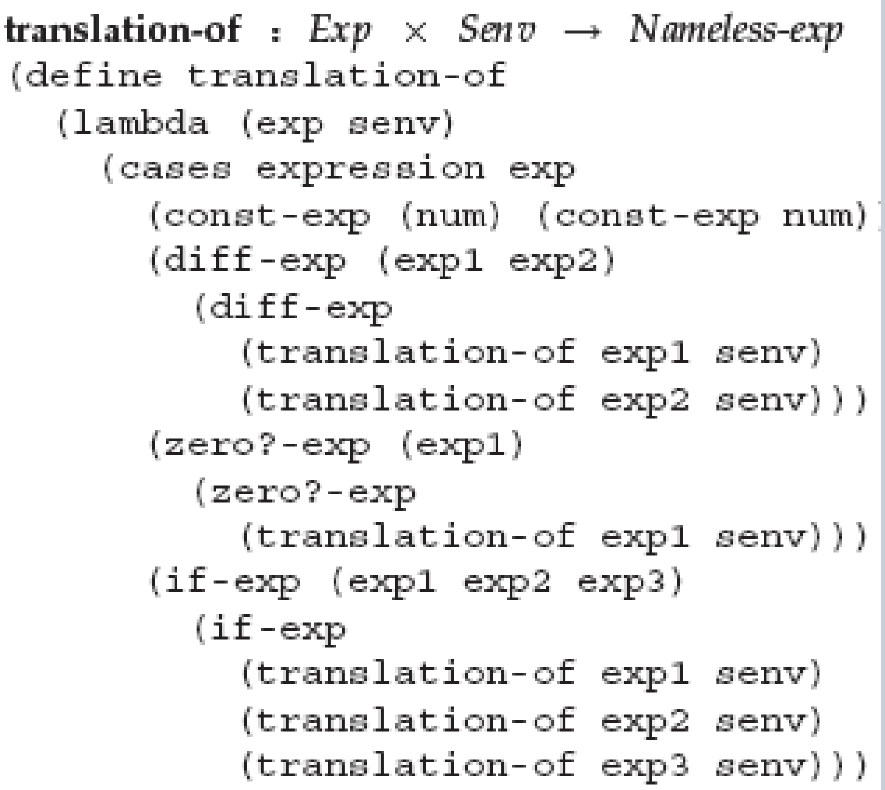
translation-of-program : *Program* → *Nameless-program*

```
(define translation-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (a-program
          (translation-of exp1 (init-senv)))))))
```

init-senv : () → *Senv*

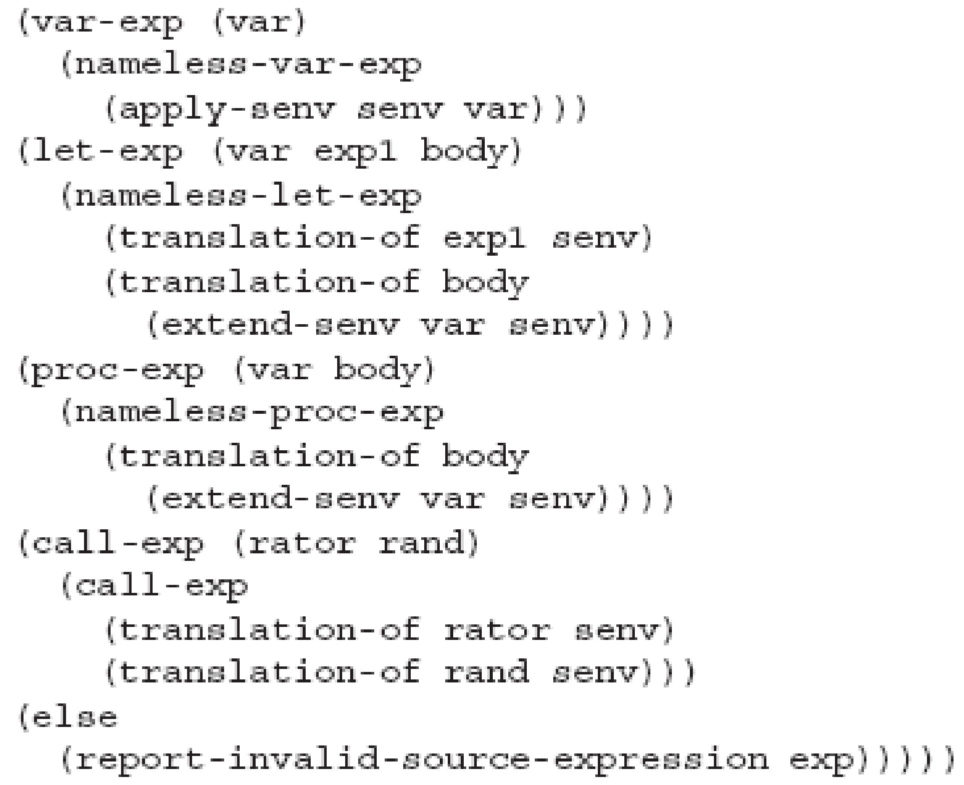
```
(define init-senv
  (lambda ()
    (extend-senv 'i
      (extend-senv 'v
        (extend-senv 'x
          (empty-senv))))))
```

Translator 2



translation-of : *Exp* × *Senv* → *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)))))
```