Announcements

1. Project 3 will be out today

Lecture 17
Scoping & Binding
Review

T. METIN SEZGIN

Denoted values

- Variables
  - References `(f x y)`
  - Declarations
```
(lambda (x) (+ x 3))
(let ((x (+ y 7))) (+ x 3))
```
- Semantics
  - Binding
  - Scope

3

What is the value of this expression?

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

4

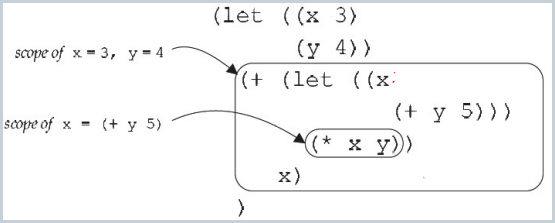
Denoted values

- Variables
  - References
  - Declarations

```
(f x y)
```

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

- Semantics
  - Binding
  - Scope



we need rules to define scoping

5

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))
```

6

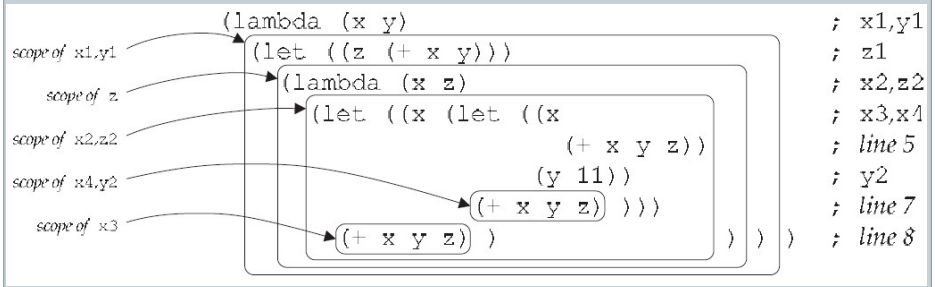
Concepts

- 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
```

7

Another example



Where are the binding rules set/defined?

8

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))
```

9

Lecture 18
Lexical Addressing and Translation

T. METIN SEZGIN

10

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

11

Nugget

Arguments to procedures always found at the expected places

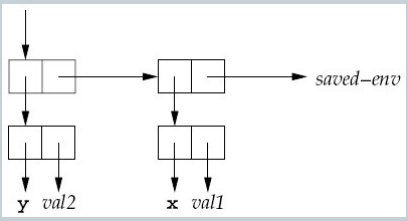
12

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)
```



13

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!

14

Nugget

We don't need names

15

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))
```

16

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))))))
```

17

Nugget

We can create a new "nameless" language

18

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 : () → 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))))))
```

21

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))))))
```

22

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)))))
```