

COSE212: Programming Languages

Lecture 7 — Design and Implementation of PLs (3) Lexical Scoping of Variables

Hakjoo Oh
2019 Fall

Goal

Understand lexical scoping in a more systematic way.

- Variable declaration and use
- Scoping rule
- Lexical address
- Nameless representation

References and Declarations

In programming languages, variables appear in two different ways:

- A variable *reference* is a use of the variable.
- A variable *declaration* introduces the variable as a name for some value.
- Examples:

`(f x y)`

`proc (x) (x + 3)`

`let x = y + 7 in x + 3`

- We say a variable reference is *bound by* the declaration with which it is associated, and that the variable is *bound to* its value.

Scoping Rules

- Every programming language has some rules to determine the corresponding declaration of a variable reference. Called *scoping rules*.
- Most programming languages use *lexical scoping* rules, where the declaration of a reference is found by searching outward from the reference until we find a declaration of the variable:

```
let x = 3                                // call this x1
  in let y = 4
    in (let x = y + 5                    // call this x2
        in x * y)                       // Here x refers to x2
      + x                                 // Here x refers to x1
```

- We can determine the declaration of each variable reference without executing the program.

Static vs. Dynamic Properties of Programs

- Properties of programs are classified into static and dynamic properties.
- Properties that can be computed without executing the program are called *static properties*.
 - ▶ ex) declaration, scope, etc
- Properties that cannot be computed without executing the program are called *dynamic properties*. Dynamic properties are only determined at run-time.
 - ▶ ex) values, types, the absence of bugs, etc.

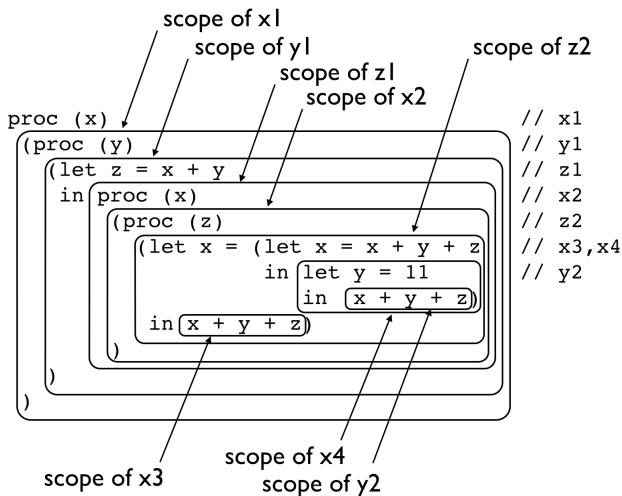
Example: Lexical Scopes of Variables

Declarations have limited *scopes*, each of which lies entirely within another:

```
proc (x)                                // x1
  (proc (y)                              // y1
    (let z = x + y                        // z1
      in proc (x)                         // x2
        (proc (z)                         // z2
          (let x = (let x = x + y + z     // x3,x4
                    in let y = 11       // y2
                      in x + y + z)
            in x + y + z)
          )
        )
      )
    )
  )
)
```

Example: Lexical Scopes of Variables

Declarations have limited *scopes*, each of which lies entirely within another:



Lexical Address

- Execution of the scoping algorithm can be viewed as a search outward from a variable reference.
- The number of declarations crossed to find the associated declaration is called the *lexical depth* of a variable reference.

```
let x = 1
  in let y = 2
    in x + y
```

- The lexical depth of a variable reference uniquely identifies the declaration to which it refers.
- Therefore, variable names are entirely removed from the program, and variable references are replaced by their *lexical address*:

```
let 1
  in let 2
    in #1 + #0
```

“Nameless” or “De Bruijn” representation.

Examples: Nameless Representation

- `(let a = 5 in proc (x) (x-a)) 7`
- `(let x = 37
 in proc (y)
 let z = (y - x)
 in (x - y)) 10`

Lexical Address

- The lexical address of a variable indicates the position of the variable in the environment.
- ```
let x = 1
 in let y = 2
 in x + y
```
- ```
(let a = 5 in proc (x) (x-a)) 7
```

Nameless Proc

Syntax

$P \rightarrow E$

$E \rightarrow n$

| $\#n$

| $E + E$

| $E - E$

| `iszero` E

| `if` E `then` E `else` E

| `let` E `in` E

| `proc` E

| $E E$

Nameless Proc

Semantics

$$\begin{aligned} \text{Val} &= \mathbb{Z} + \text{Bool} + \text{Procedure} \\ \text{Procedure} &= \mathbf{E} \times \text{Env} \\ \text{Env} &= \text{Val}^* \end{aligned}$$

$$\frac{}{\rho \vdash n \Rightarrow n} \quad \frac{}{\rho \vdash \#n \Rightarrow \rho n} \quad \frac{\rho \vdash E_1 \Rightarrow n_1 \quad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 + E_2 \Rightarrow n_1 + n_2}$$

$$\frac{\rho \vdash E \Rightarrow 0}{\rho \vdash \text{iszero } E \Rightarrow \text{true}} \quad \frac{\rho \vdash E \Rightarrow n}{\rho \vdash \text{iszero } E \Rightarrow \text{false}} \quad n \neq 0$$

$$\frac{\rho \vdash E_1 \Rightarrow \text{true} \quad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \Rightarrow v} \quad \frac{\rho \vdash E_1 \Rightarrow \text{false} \quad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \Rightarrow v}$$

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \quad v_1 :: \rho \vdash E_2 \Rightarrow v}{\rho \vdash \text{let } E_1 \text{ in } E_2 \Rightarrow v}$$

$$\frac{}{\rho \vdash \text{proc } E \Rightarrow (E, \rho)}$$

$$\frac{\rho \vdash E_1 \Rightarrow (E, \rho') \quad \rho \vdash E_2 \Rightarrow v \quad v :: \rho' \vdash E \Rightarrow v'}{\rho \vdash E_1 E_2 \Rightarrow v'}$$

Example

$\square \vdash (\text{let } 37 \text{ in proc } (\text{let } (\#0 \text{ -}\#1) \text{ in } (\#2 \text{ - } \#1))) 10 \Rightarrow \mathbf{27}$

Translation

The nameless version of a program P is defined to be $\mathbf{trans}(E)(\rho)$:

$$\begin{aligned}\mathbf{trans}(n)(\rho) &= n \\ \mathbf{trans}(x)(\rho) &= \#n \quad (n \text{ is the first position of } x \text{ in } \rho) \\ \mathbf{trans}(E_1 + E_2)(\rho) &= \mathbf{trans}(E_1)(\rho) + \mathbf{trans}(E_2)(\rho) \\ \mathbf{trans}(\text{iszero } E)(\rho) &= \text{iszero } (\mathbf{trans}(E)(\rho)) \\ \mathbf{trans}(\text{if } E_1 \text{ then } E_2 \text{ else } E_3)(\rho) &= \text{if } \mathbf{trans}(E_1)(\rho) \\ &\quad \text{then } \mathbf{trans}(E_2)(\rho) \text{ else } \mathbf{trans}(E_3)(\rho) \\ \mathbf{trans}(\text{let } x = E_1 \text{ in } E_2)(\rho) &= \text{let } \mathbf{trans}(E_1)(\rho) \text{ in } \mathbf{trans}(E_2)(x :: \rho) \\ \mathbf{trans}(\text{proc}(x) E)(\rho) &= \text{proc } \mathbf{trans}(E)(x :: \rho) \\ \mathbf{trans}(E_1 E_2)(\rho) &= \mathbf{trans}(E_1)(\rho) \mathbf{trans}(E_2)(\rho)\end{aligned}$$

Example

$$\mathbf{trans} \left(\begin{array}{l} (\text{let } x = 37 \\ \text{in proc } (y) \\ \text{let } z = (y - x) \\ \text{in } (x - y)) \ 10 \end{array} \right) ([\]) =$$

Summary

- In lexical scoping, scoping rules are static properties: nameless representation with lexical addresses.
- Lexical address predicts the place of the variable in the environment.
- Compilers routinely use the nameless representation: Given an input program P ,
 - 1 translate it to $\mathbf{trans}(P)([])$,
 - 2 execute the nameless program.