Lecture 7 — Design and Implementation of PLs

(3) Lexical Scoping of Variables

CSE307: Programming Languages

Minseok Jeon

2026 Spring

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.

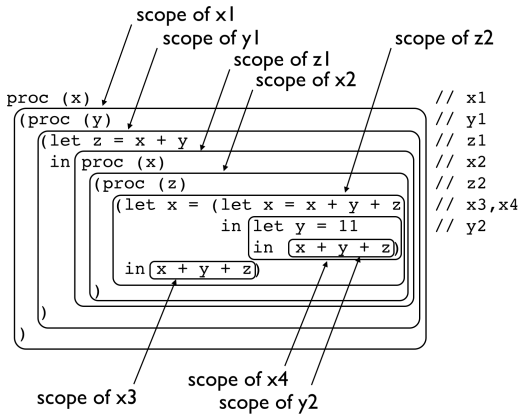
Example: Lexical Scopes of Variables

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 identifes 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 5 in proc (#0 - #1)) 7`

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

  ```
  (let 37
   in proc
       let (#0 - #1)
       in (#2 - #1)) 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

$$
\begin{array}{lcl}
P & \to & E \\
E & \to & n \\
 & | & \#n \\
 & | & E + E \\
 & | & E - E \\
 & | & \texttt{iszero } E \\
 & | & \texttt{if } E \texttt{ then } E \texttt{ else } E \\
 & | & \texttt{let } E \texttt{ in } E \\
 & | & \texttt{proc } E \\
 & | & E\ E
\end{array}
$$

Nameless Proc

Semantics

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

$$\frac{}{\boldsymbol{\rho} \vdash \boldsymbol{n} \Rightarrow \boldsymbol{n}} \qquad \frac{}{\boldsymbol{\rho} \vdash \#\boldsymbol{n} \Rightarrow \boldsymbol{\rho}_{\boldsymbol{n}}} \qquad \frac{\boldsymbol{\rho} \vdash \boldsymbol{E}_1 \Rightarrow \boldsymbol{n}_1 \qquad \boldsymbol{\rho} \vdash \boldsymbol{E}_2 \Rightarrow \boldsymbol{n}_2}{\boldsymbol{\rho} \vdash \boldsymbol{E}_1 + \boldsymbol{E}_2 \Rightarrow \boldsymbol{n}_1 + \boldsymbol{n}_2}$$

$$\frac{\boldsymbol{\rho} \vdash \boldsymbol{E} \Rightarrow 0}{\boldsymbol{\rho} \vdash \texttt{iszero}\ \boldsymbol{E} \Rightarrow \mathit{true}} \qquad \frac{\boldsymbol{\rho} \vdash \boldsymbol{E} \Rightarrow \boldsymbol{n}}{\boldsymbol{\rho} \vdash \texttt{iszero}\ \boldsymbol{E} \Rightarrow \mathit{false}}\ \boldsymbol{n} \neq 0$$

$$\frac{\boldsymbol{\rho} \vdash \boldsymbol{E}_1 \Rightarrow \mathit{true} \qquad \boldsymbol{\rho} \vdash \boldsymbol{E}_2 \Rightarrow \boldsymbol{v}}{\boldsymbol{\rho} \vdash \texttt{if}\ \boldsymbol{E}_1\ \texttt{then}\ \boldsymbol{E}_2\ \texttt{else}\ \boldsymbol{E}_3 \Rightarrow \boldsymbol{v}} \qquad \frac{\boldsymbol{\rho} \vdash \boldsymbol{E}_1 \Rightarrow \mathit{false} \qquad \boldsymbol{\rho} \vdash \boldsymbol{E}_3 \Rightarrow \boldsymbol{v}}{\boldsymbol{\rho} \vdash \texttt{if}\ \boldsymbol{E}_1\ \texttt{then}\ \boldsymbol{E}_2\ \texttt{else}\ \boldsymbol{E}_3 \Rightarrow \boldsymbol{v}}$$

$$\frac{\boldsymbol{\rho} \vdash \boldsymbol{E}_1 \Rightarrow \boldsymbol{v}_1 \qquad \boldsymbol{v}_1 :: \boldsymbol{\rho} \vdash \boldsymbol{E}_2 \Rightarrow \boldsymbol{v}}{\boldsymbol{\rho} \vdash \texttt{let}\ \boldsymbol{E}_1\ \texttt{in}\ \boldsymbol{E}_2 \Rightarrow \boldsymbol{v}}$$

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

$$\frac{\boldsymbol{\rho} \vdash \boldsymbol{E}_1 \Rightarrow (\boldsymbol{E}, \boldsymbol{\rho}') \qquad \boldsymbol{\rho} \vdash \boldsymbol{E}_2 \Rightarrow \boldsymbol{v} \qquad \boldsymbol{v} :: \boldsymbol{\rho}' \vdash \boldsymbol{E} \Rightarrow \boldsymbol{v}'}{\boldsymbol{\rho} \vdash \boldsymbol{E}_1\ \boldsymbol{E}_2 \Rightarrow \boldsymbol{v}'}$$

Example

$$[] \vdash \texttt{(let 37 in proc (let (\#0 -\#1) in (\#2 - \#1))) 10} \Rightarrow 27$$

Example

$\boldsymbol{E_b} = \texttt{let}\ (\#0 - \#1)\ \texttt{in}\ (\#2 - \#1)$.

$\mathcal{D}_2$: (let rule, with env $[10, 37]$)

$$\dfrac{\dfrac{\overline{[10,37] \vdash \#0 \Rightarrow 10} \quad \overline{[10,37] \vdash \#1 \Rightarrow 37}}{[10,37] \vdash \#0 - \#1 \Rightarrow -27} \quad \dfrac{\overline{[-27,10,37] \vdash \#2 \Rightarrow 37} \quad \overline{[-27,10,37] \vdash \#1 \Rightarrow 10}}{[-27,10,37] \vdash \#2 - \#1 \Rightarrow 27}}{[10,37] \vdash \boldsymbol{E_b} \Rightarrow 27}$$

$\mathcal{D}_1$: (let + proc)

$$\dfrac{\overline{[] \vdash 37 \Rightarrow 37} \qquad \overline{[37] \vdash \texttt{proc}\ \boldsymbol{E_b} \Rightarrow (\boldsymbol{E_b}, [37])}}{[] \vdash \texttt{let}\ 37\ \texttt{in proc}\ \boldsymbol{E_b} \Rightarrow (\boldsymbol{E_b}, [37])}$$

Main: (application rule)

$$\dfrac{\mathcal{D}_1 \qquad \overline{[] \vdash 10 \Rightarrow 10} \qquad \mathcal{D}_2}{[] \vdash (\texttt{let}\ 37\ \texttt{in proc}\ \boldsymbol{E_b})\ 10 \Rightarrow 27}$$

Translation

The nameless version of a program $\boldsymbol{P}$ is defined to be $\textbf{trans}(\boldsymbol{E})([])$:

$$
\begin{array}{rcl}
\textbf{trans}(n)(\rho) & = & n \\
\textbf{trans}(x)(\rho) & = & \#n \qquad (n \text{ is the first position of } x \text{ in } \rho) \\
\textbf{trans}(E_1 + E_2)(\rho) & = & \textbf{trans}(E_1)(\rho) + \textbf{trans}(E_2)(\rho) \\
\textbf{trans}(\texttt{iszero}\, E)(\rho) & = & \texttt{iszero}\ (\textbf{trans}(E)(\rho)) \\
\textbf{trans}(\texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3)(\rho) & = & \texttt{if}\ \textbf{trans}(E_1)(\rho) \\
 & & \texttt{then}\ \textbf{trans}(E_2)(\rho)\ \texttt{else}\ \textbf{trans}(E_3)(\rho) \\
\textbf{trans}(\texttt{let}\ x = E_1\ \texttt{in}\ E_2)(\rho) & = & \texttt{let}\ \textbf{trans}(E_1)(\rho)\ \texttt{in}\ \textbf{trans}(E_2)(x :: \rho) \\
\textbf{trans}(\texttt{proc}(x)\ E)(\rho) & = & \texttt{proc}\ \textbf{trans}(E)(x :: \rho) \\
\textbf{trans}(E_1\ E_2)(\rho) & = & \textbf{trans}(E_1)(\rho)\ \textbf{trans}(E_2)(\rho)
\end{array}
$$

Example

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

Example

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

$= \textbf{trans}(\texttt{let x=37 in proc (y) ...})([])\ \ \textbf{trans}(\texttt{10})([])$

$= (\texttt{let}\ \textbf{trans}(\texttt{37})([])\ \texttt{in}\ \textbf{trans}(\texttt{proc (y) ...})([\boldsymbol{x}]))\ \ \texttt{10}$

$= (\texttt{let 37 in proc}\ \textbf{trans}(\texttt{let z=(y-x) in (x-y)})([\boldsymbol{y}, \boldsymbol{x}]))\ \ \texttt{10}$

$= (\texttt{let 37 in proc (let}\ \textbf{trans}(\texttt{y-x})([\boldsymbol{y}, \boldsymbol{x}])\ \texttt{in}\ \textbf{trans}(\texttt{x-y})([z, \boldsymbol{y}, \boldsymbol{x}])))\ \ \texttt{10}$

$= (\texttt{let 37 in proc (let (\#0 - \#1) in (\#2 - \#1)))}\ \ \texttt{10}$

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 $\boldsymbol{P}$,
  1. translate it to $\textbf{trans}(\boldsymbol{P})([])$,
  2. execute the nameless program.