ENE4014: Programming Languages

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

Woosuk Lee
2024 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.

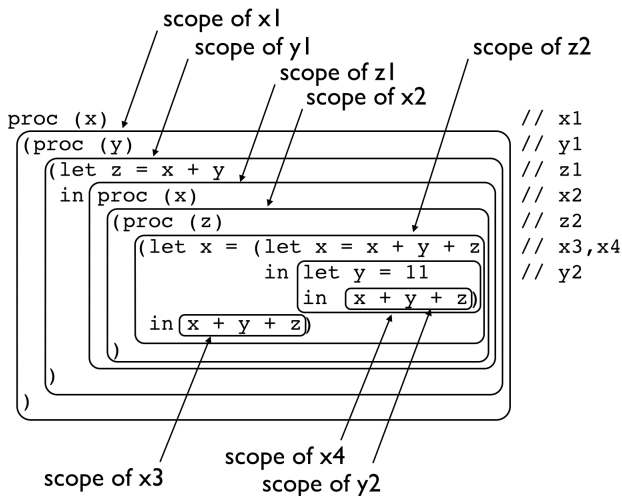# Example: Lexical Scopes of Variables

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 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

$$
\begin{aligned}
P &\rightarrow E \\
E &\rightarrow n \\
&\mid\ \#n \\
&\mid\ E + E \\
&\mid\ E - E \\
&\mid\ \texttt{iszero } E \\
&\mid\ \texttt{if } E \texttt{ then } E \texttt{ else } E \\
&\mid\ \texttt{let } E \texttt{ in } E \\
&\mid\ \texttt{proc } E \\
&\mid\ E\ E
\end{aligned}
$$

## Nameless Proc

Semantics

$$
\begin{array}{rcl}
Val &=& \mathbb{Z} + Bool + Procedure \\
Procedure &=& E \times Env \\
Env &=& Val^*
\end{array}
$$

$$\overline{\rho \vdash n \Rightarrow n} \qquad \overline{\rho \vdash \#n \Rightarrow \rho_n} \qquad \dfrac{\rho \vdash E_1 \Rightarrow n_1 \quad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 + E_2 \Rightarrow n_1 + n_2}$$

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

$$\dfrac{\rho \vdash E_1 \Rightarrow true \quad \rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v} \qquad \dfrac{\rho \vdash E_1 \Rightarrow false \quad \rho \vdash E_3 \Rightarrow v}{\rho \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 \Rightarrow v}$$

$$\dfrac{\rho \vdash E_1 \Rightarrow v_1 \quad v_1 :: \rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{let } E_1 \texttt{ in } E_2 \Rightarrow v}$$

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

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

# Example

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

## Translation

The nameless version of a program $P$ is defined to be **trans**$(E)([])$:

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

# Example

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

# 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 **trans**$(P)([])$,
  2. execute the nameless program.