# ENE4014: Programming Languages

# Lecture 8 — Design and Implementation of PLs (4) States

Woosuk Lee
2026 Spring

# Review: Our Language So Far

Our language has expressions and procedures.

Syntax

$$
\begin{array}{rcl}
\boldsymbol{P} & \rightarrow & \boldsymbol{E} \\
\boldsymbol{E} & \rightarrow & \boldsymbol{n} \\
 & | & \boldsymbol{x} \\
 & | & \boldsymbol{E} + \boldsymbol{E} \\
 & | & \boldsymbol{E} - \boldsymbol{E} \\
 & | & \texttt{iszero } \boldsymbol{E} \\
 & | & \texttt{if } \boldsymbol{E} \texttt{ then } \boldsymbol{E} \texttt{ else } \boldsymbol{E} \\
 & | & \texttt{let } \boldsymbol{x} = \boldsymbol{E} \texttt{ in } \boldsymbol{E} \\
 & | & \texttt{read} \\
 & | & \texttt{letrec } \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{E} \texttt{ in } \boldsymbol{E} \\
 & | & \texttt{proc } \boldsymbol{x}\ \boldsymbol{E} \\
 & | & \boldsymbol{E}\ \boldsymbol{E}
\end{array}
$$

# Review: Our Language So Far

Semantics

$$\frac{}{\rho \vdash n \Rightarrow n} \qquad \frac{}{\rho \vdash x \Rightarrow \rho(x)} \qquad \frac{\rho \vdash E_1 \Rightarrow n_1 \qquad \rho \vdash E_2 \Rightarrow n_2}{\rho \vdash E_1 + E_2 \Rightarrow n_1 + n_2}$$

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

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

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \qquad [x \mapsto v_1]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 \Rightarrow v} \qquad \frac{[f \mapsto (f, x, E_1, \rho)]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \texttt{letrec}\ f(x) = E_1\ \texttt{in}\ E_2 \Rightarrow v}$$

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

$$\frac{\rho \vdash E_1 \Rightarrow (x, E, \rho') \qquad \rho \vdash E_2 \Rightarrow v \qquad [x \mapsto v]\rho' \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

$$\frac{\rho \vdash E_1 \Rightarrow (f, x, E, \rho') \quad \rho \vdash E_2 \Rightarrow v \quad [x \mapsto v, f \mapsto (f, x, E, \rho')]\rho' \vdash E \Rightarrow v'}{\rho \vdash E_1\ E_2 \Rightarrow v'}$$

# This Lecture: Adding States to the Language

- So far, our language only had the values produced by computation.
- But computation also has *effects*: it may change the state of memory.
- We will extend the language to support computational effects:
  - Syntax for creating and using memory locations
  - Semantics for manipulating memory states

# Motivating Example

- How can we compute the number of times `f` has been called?

```
let f = proc (x) (x)
in (f (f 1))
```

# Motivating Example

- How can we compute the number of times `f` has been called?

```
let f = proc (x) (x)
in (f (f 1))
```

- Does the following program work?

```
let counter = 0
in let f = proc (x) (let counter = counter + 1
                     in x)
   in let a = (f (f 1))
      in counter
```

# Motivating Example

- How can we compute the number of times `f` has been called?

```
let f = proc (x) (x)
in (f (f 1))
```

- Does the following program work?

```
let counter = 0
in let f = proc (x) (let counter = counter + 1
                     in x)
   in let a = (f (f 1))
      in counter
```

- The binding of `counter` is local. We need global *effects*.
- Effects are implemented by introducing *memory* (*store*) and *locations* (*reference*).

# Two Approaches

Programming languages support references explicitly or implicitly.

- Languages with explicit references provide a clear account of allocation, dereference, and mutation of memory cells.
  - e.g., OCaml, F#
- In languages with implicit references, references are built-in. References are not explicitly manipulated.
  - e.g., C and Java.

# A Language with Explicit References

$$
\begin{array}{lcl}
P & \to & E \\
E & \to & n \mid x \\
 & \mid & E + E \mid E - E \\
 & \mid & \texttt{iszero}\ E \mid \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\
 & \mid & \texttt{let}\ x = E\ \texttt{in}\ E \\
 & \mid & \texttt{proc}\ x\ E \mid E\ E \\
 & \mid & \texttt{ref}\ E \\
 & \mid & !\ E \\
 & \mid & E := E \\
 & \mid & E; E
\end{array}
$$

- $\texttt{ref}\ E$ allocates a new location, store the value of $E$ in it, and returns it.
- $!\ E$ returns the contents of the location that $E$ refers to.
- $E_1 := E_2$ changes the contents of the location ($E_1$) by the value of $E_2$.
- $E_1; E_2$ executes $E_1$ and then $E_2$ while accumulating effects.

# Example 1

- ```
  let counter = ref 0
  in let f = proc (x) (counter := !counter + 1; !counter)
     in let a = (f 0)
        in let b = (f 0)
           in (a - b)
  ```
  ```

# Example 1

- ```
  let counter = ref 0
  in let f = proc (x) (counter := !counter + 1; !counter)
     in let a = (f 0)
        in let b = (f 0)
           in (a - b)
  ```
- ```
  let f = let counter = ref 0
          in proc (x) (counter := !counter + 1; !counter)
  in let a = (f 0)
     in let b = (f 0)
        in (a - b)
  ```

# Example 1

- ```
  let counter = ref 0
  in let f = proc (x) (counter := !counter + 1; !counter)
     in let a = (f 0)
        in let b = (f 0)
           in (a - b)
  ```
- ```
  let f = let counter = ref 0
          in proc (x) (counter := !counter + 1; !counter)
  in let a = (f 0)
     in let b = (f 0)
        in (a - b)
  ```
- ```
  let f = proc (x) (let counter = ref 0
                    in (counter := !counter + 1; !counter))
  in let a = (f 0)
     in let b = (f 0)
        in (a - b)
  ```
  ```

# Example 2

We can make chains of references:

```
let x = ref (ref 0)
in (!x := 11; !(!x))
```

# Semantics

Memory is modeled as a finite map from locations to values:

$$\begin{aligned} \boldsymbol{Val} &= \mathbb{Z} + \boldsymbol{Bool} + \boldsymbol{Procedure} + \boldsymbol{Loc} \\ \boldsymbol{Procedure} &= \boldsymbol{Var} \times \boldsymbol{E} \times \boldsymbol{Env} \\ \boldsymbol{\rho} \in \boldsymbol{Env} &= \boldsymbol{Var} \to \boldsymbol{Val} \\ \boldsymbol{\sigma} \in \boldsymbol{Mem} &= \boldsymbol{Loc} \to \boldsymbol{Val} \end{aligned}$$

Semantics rules additionally describe memory effects:

$$\boldsymbol{\rho}, \boldsymbol{\sigma} \vdash \boldsymbol{E} \Rightarrow \boldsymbol{v}, \boldsymbol{\sigma}'$$

# Semantics

Existing rules are enriched with memory effects:

$$\frac{}{\rho, \sigma \vdash n \Rightarrow n, \sigma} \qquad \frac{}{\rho, \sigma \vdash x \Rightarrow \rho(x), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow n_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow n_2, \sigma_2}{\rho, \sigma_0 \vdash E_1 + E_2 \Rightarrow n_1 + n_2, \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow 0, \sigma_1}{\rho, \sigma_0 \vdash \texttt{iszero}\ E \Rightarrow \mathit{true}, \sigma_1} \qquad \frac{\rho, \sigma_0 \vdash E \Rightarrow n, \sigma_1}{\rho, \sigma_0 \vdash \texttt{iszero}\ E \Rightarrow \mathit{false}, \sigma_1}\ n \neq 0$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow \mathit{true}, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v, \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow \mathit{false}, \sigma_1 \qquad \rho, \sigma_1 \vdash E_3 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v, \sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad [x \mapsto v_1]\rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 \Rightarrow v, \sigma_2}$$

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

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \qquad [x \mapsto v]\rho', \sigma_2 \vdash E \Rightarrow v', \sigma_3}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}$$

# Semantics

Rules for new constructs:

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash \texttt{ref}\ E \Rightarrow l, [l \mapsto v]\sigma_1} \quad l \notin \mathrm{Dom}(\sigma_1)$$

$$\frac{\rho, \sigma_0 \vdash E \Rightarrow l, \sigma_1}{\rho, \sigma_0 \vdash !\ E \Rightarrow \sigma_1(l), \sigma_1}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow l, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash E_1 := E_2 \Rightarrow v, [l \mapsto v]\sigma_2}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2}{\rho, \sigma_0 \vdash E_1; E_2 \Rightarrow v_2, \sigma_2}$$

# Example

$\boldsymbol{\rho}, \boldsymbol{\sigma_0} \vdash$ `let x = ref (ref 0) in (!x := 11; !(!x))` $\Rightarrow$

# Exercise

Extend the language with recursive procedures:

$$
\begin{array}{rcl}
P & \rightarrow & E \\
E & \rightarrow & n \mid x \\
 & \mid & E + E \mid E - E \\
 & \mid & \texttt{iszero } E \mid \texttt{if } E \texttt{ then } E \texttt{ else } E \\
 & \mid & \texttt{let } x = E \texttt{ in } E \\
 & \mid & \texttt{letrec } f(x) = E \texttt{ in } E \\
 & \mid & \texttt{proc } x\ E \mid E\ E \\
 & \mid & \texttt{ref } E \\
 & \mid & !\ E \\
 & \mid & E := E \\
 & \mid & E; E
\end{array}
$$

# Exercise (Continued)

- Domain:

$$\begin{aligned} \boldsymbol{Val} &= \boldsymbol{\ldots + RecProcedure} \\ \boldsymbol{RecProcedure} &= \boldsymbol{Var \times Var \times E \times Env} \\ \boldsymbol{\rho \in Env} &= \boldsymbol{Var \to Val} \\ \boldsymbol{\sigma \in Mem} &= \boldsymbol{Loc \to Val} \end{aligned}$$

- Semantics rules:

$$\frac{}{\boldsymbol{\rho, \sigma_0 \vdash \texttt{letrec } f(x) = E_1 \texttt{ in } E_2 \Rightarrow}}$$

$$\frac{}{\boldsymbol{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow}}$$

# A Language with Implicit References

$$
\begin{array}{lcl}
P & \rightarrow & E \\
E & \rightarrow & n \mid x \\
 & \mid & E + E \mid E - E \\
 & \mid & \texttt{iszero}\ E \mid \texttt{if}\ E\ \texttt{then}\ E\ \texttt{else}\ E \\
 & \mid & \texttt{let}\ x = E\ \texttt{in}\ E \\
 & \mid & \texttt{proc}\ x\ E \mid E\ E \\
 & \mid & x := E \\
 & \mid & E; E
\end{array}
$$

- In this design, every variable denotes a reference and is mutable.
- $x := E$ changes the contents of $x$ by the value of $E$.

# Examples

Computing the number of times `f` has been called:

- ```
  let counter = 0
  in let f = proc (x) (counter := counter + 1; counter)
     in let a = (f 0)
        in let b = (f 0)
           in (a-b)
  ```
  ```

# Examples

Computing the number of times `f` has been called:

- ```
  let counter = 0
  in let f = proc (x) (counter := counter + 1; counter)
     in let a = (f 0)
        in let b = (f 0)
           in (a-b)
  ```
- ```
  let f = let counter = 0
          in proc (x) (counter := counter + 1; counter)
  in let a = (f 0)
     in let b = (f 0)
        in (a-b)
  ```

# Examples

Computing the number of times `f` has been called:

- ```
  let counter = 0
  in let f = proc (x) (counter := counter + 1; counter)
     in let a = (f 0)
        in let b = (f 0)
           in (a-b)
  ```
- ```
  let f = let counter = 0
          in proc (x) (counter := counter + 1; counter)
  in let a = (f 0)
     in let b = (f 0)
        in (a-b)
  ```
- ```
  let f = proc (x) (let counter = 0
                    in (counter := counter + 1; counter))
  in let a = (f 0)
     in let b = (f 0)
        in (a-b)
  ```
  ```

# Exercise

What is the result of the program?

```
let f = proc (x)
          proc (y)
            (x := x + 1; x - y)
in ((f 44) 33)
```

# Semantics

References are no longer values and every variable denotes a reference:

$$
\begin{aligned}
\boldsymbol{Val} &= \mathbb{Z} + \boldsymbol{Bool} + \boldsymbol{Procedure} \\
\boldsymbol{Procedure} &= \boldsymbol{Var} \times \boldsymbol{E} \times \boldsymbol{Env} \\
\boldsymbol{\rho} \in \boldsymbol{Env} &= \boldsymbol{Var} \to \boldsymbol{Loc} \\
\boldsymbol{\sigma} \in \boldsymbol{Mem} &= \boldsymbol{Loc} \to \boldsymbol{Val}
\end{aligned}
$$

# Semantics

$$\frac{}{\rho, \sigma \vdash n \Rightarrow n, \sigma} \qquad \frac{}{\rho, \sigma \vdash x \Rightarrow \sigma(\rho(x)), \sigma}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow n_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow n_2, \sigma_2}{\rho, \sigma_0 \vdash E_1 + E_2 \Rightarrow n_1 + n_2, \sigma_2} \qquad \frac{\rho, \sigma_0 \vdash E \Rightarrow 0, \sigma_1}{\rho, \sigma_0 \vdash \texttt{iszero}\ E \Rightarrow true, \sigma_1}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow true, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{if}\ E_1\ \texttt{then}\ E_2\ \texttt{else}\ E_3 \Rightarrow v, \sigma_2}$$

$$\frac{}{\rho, \sigma \vdash \texttt{proc}\ x\ E \Rightarrow (x, E, \rho), \sigma} \qquad \frac{\rho, \sigma_0 \vdash E \Rightarrow v, \sigma_1}{\rho, \sigma_0 \vdash x := E \Rightarrow v, [\rho(x) \mapsto v]\sigma_1}$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad [x \mapsto l]\rho, [l \mapsto v_1]\sigma_1 \vdash E_2 \Rightarrow v, \sigma_2}{\rho, \sigma_0 \vdash \texttt{let}\ x = E_1\ \texttt{in}\ E_2 \Rightarrow v, \sigma_2}\ l \notin \mathrm{Dom}(\sigma_1)$$

$$\frac{\begin{array}{c}\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \\ [x \mapsto l]\rho', [l \mapsto v]\sigma_2 \vdash E \Rightarrow v', \sigma_3\end{array}}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}\ l \notin \mathrm{Dom}(\sigma_2)$$

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow v_1, \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v_2, \sigma_2}{\rho, \sigma_0 \vdash E_1; E_2 \Rightarrow v_2, \sigma_2}$$

# Example

```
let f = let count = 0
        in proc (x) (count := count + 1; count)
in let a = (f 0)
   in let b = (f 0)
      in a - b
```

# Exercise

Extend the language with recursive procedures:

$$
\begin{array}{rcl}
P & \rightarrow & E \\
E & \rightarrow & n \mid x \\
 & \mid & E + E \mid E - E \\
 & \mid & \texttt{iszero } E \mid \texttt{if } E \texttt{ then } E \texttt{ else } E \\
 & \mid & \texttt{let } x = E \texttt{ in } E \\
 & \mid & \texttt{letrec } f(x) = E \texttt{ in } E \\
 & \mid & \texttt{proc } x\ E \mid E\ E \\
 & \mid & x := E \\
 & \mid & E; E
\end{array}
$$

# Exercise (Continued)

- Domain:

$$\begin{aligned} \boldsymbol{Val} &= \boldsymbol{\ldots + RecProcedure} \\ \boldsymbol{RecProcedure} &= \boldsymbol{Var \times Var \times E \times Env} \\ \boldsymbol{\rho \in Env} &= \boldsymbol{Var \to Loc} \\ \boldsymbol{\sigma \in Mem} &= \boldsymbol{Loc \to Val} \end{aligned}$$

- Semantics rules:

$$\frac{}{\boldsymbol{\rho, \sigma_0 \vdash \texttt{letrec } f(x) = E_1 \texttt{ in } E_2 \Rightarrow}}$$

$$\frac{}{\boldsymbol{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow}}$$

# Parameter-Passing Variations

- Our current strategy of calling a procedure is *call-by-value*. The formal parameter refers to a new location containing the value of the actual parameter:

$$\frac{\boldsymbol{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1} \qquad \boldsymbol{\rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2} \qquad \boldsymbol{[x \mapsto l]\rho', [l \mapsto v]\sigma_2 \vdash E \Rightarrow v', \sigma_3}}{\boldsymbol{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}} \; l \notin \mathrm{Dom}(\boldsymbol{\sigma_2})$$

- The most commonly used form of parameter-passing.

# Parameter-Passing Variations

- Our current strategy of calling a procedure is *call-by-value*. The formal parameter refers to a new location containing the value of the actual parameter:

$$\frac{\boldsymbol{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1} \qquad \boldsymbol{\rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2} \qquad \boldsymbol{[x \mapsto l]\rho', [l \mapsto v]\sigma_2 \vdash E \Rightarrow v', \sigma_3}}{\boldsymbol{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}} \; \boldsymbol{l \notin} \text{Dom}(\boldsymbol{\sigma_2})$$

- The most commonly used form of parameter-passing.
- For example, the assignment to `x` has no effect on the contents of `a`:

```
let p = proc (x) (x := 4)
in let a = 3
   in ((p a); a)
```

# Parameter-Passing Variations

- Our current strategy of calling a procedure is *call-by-value*. The formal parameter refers to a new location containing the value of the actual parameter:

$$\frac{\rho, \sigma_0 \vdash E_1 \Rightarrow (x, E, \rho'), \sigma_1 \qquad \rho, \sigma_1 \vdash E_2 \Rightarrow v, \sigma_2 \qquad [x \mapsto l]\rho', [l \mapsto v]\sigma_2 \vdash E \Rightarrow v', \sigma_3}{\rho, \sigma_0 \vdash E_1\ E_2 \Rightarrow v', \sigma_3}\ l \notin \mathrm{Dom}(\sigma_2)$$

- The most commonly used form of parameter-passing.
- For example, the assignment to `x` has no effect on the contents of `a`:

```
let p = proc (x) (x := 4)
in let a = 3
   in ((p a); a)
```

- Under *call-by-reference*, the assignment changes the value of `a` after the call.

# Call-By-Reference Parameter-Passing

The location of the caller's variable is passed, rather than the contents of the variable.

- Extend the syntax:

$$\begin{array}{rcl} \boldsymbol{E} & \rightarrow & \vdots \\ & | & \boldsymbol{E}\ \boldsymbol{E} \\ & | & \boldsymbol{E}\ \langle \boldsymbol{y} \rangle \end{array}$$

- Extend the semantics:

$$\frac{\boldsymbol{\rho}, \boldsymbol{\sigma_0} \vdash \boldsymbol{E_1} \Rightarrow (\boldsymbol{x}, \boldsymbol{E}, \boldsymbol{\rho'}), \boldsymbol{\sigma_1} \qquad [\boldsymbol{x} \mapsto \boldsymbol{\rho}(\boldsymbol{y})]\boldsymbol{\rho'}, \boldsymbol{\sigma_1} \vdash \boldsymbol{E} \Rightarrow \boldsymbol{v'}, \boldsymbol{\sigma_2}}{\boldsymbol{\rho}, \boldsymbol{\sigma_0} \vdash \boldsymbol{E_1}\ \langle \boldsymbol{y} \rangle \Rightarrow \boldsymbol{v'}, \boldsymbol{\sigma_2}}$$

What is the benefit of call-by-reference compared to call-by-value?

# Examples

- ```
  let p = proc (x) (x := 4)
  in let a = 3
     in ((p <a>); a)
  ```
  ```

# Examples

- ```
  let p = proc (x) (x := 4)
  in let a = 3
     in ((p <a>); a)
  ```
- ```
  let f = proc (x) (x := 44)
  in let g = proc (y) (f <y>)
     in let z = 55
        in ((g <z>); z)
  ```

# Examples

- ```
  let p = proc (x) (x := 4)
  in let a = 3
     in ((p <a>); a)
  ```
- ```
  let f = proc (x) (x := 44)
  in let g = proc (y) (f <y>)
     in let z = 55
        in ((g <z>); z)
  ```
- ```
  let swap = proc (x) proc (y)
               let temp = x
               in (x := y; set y = temp)
  in let a = 33
     in let b = 44
        in (((swap <a>) <b>); (a-b))
  ```
  ```

# Variable Aliasing

More than one call-by-reference parameter may refer to the same location:

```
let b = 3
in let p = proc (x) proc (y)
             (x := 4; y)
   in ((p <b>) <b>)
```

- A *variable aliasing* is created: `x` and `y` refer to the same location
- With aliasing, reasoning about program behavior is very difficult, because an assignment to one variable may change the value of another.

# Lazy Evaluation

- So far all the parameter-passing strategies are *eager* in that they always evaluate the actual parameter before calling a procedure.
- In eager evaluation, procedure arguments are completely evaluated before passing them to the procedure.
- On the other hand, *lazy evaluation* delays the evaluation of arguments until it is actually needed. If the procedure body never uses the parameter, it will never be evaluated.
- Lazy evaluation potentially avoids non-termination:

```
letrec infinite(x) = (infinite x)
in let f = proc (x) (1)
   in  (f (infinite 0))
```

# Lazy Evaluation

Comparison to Eager evaluation

| Eager Evaluation | Lazy Evaluation |
|---|---|
| Space-efficient | Time-efficient |
| Cannot avoid non-termination | Can avoid non-termination |
| Easy to reason with the order of evaluation | Hard |
| Easy to reason with programs with effects | Hard |

# Summary

Our language is now (somewhat) realistic:

- expressions, procedures, recursion,
- states with explicit/implicit references
- parameter-passing variations