

ENE4014: Programming Languages

Lecture 6 — Design and Implementation of PLs (2) Procedures

Woosuk Lee
2024 Spring

Review: The Let Language

Syntax:

$P \rightarrow E$

$E \rightarrow n$

| x

| $E + E$

| $E - E$

| iszero E

| if E then E else E

| let $x = E$ in E

| read

Review: The Let Language

Semantic domain:

$$\begin{aligned} \mathit{Val} &= \mathbb{Z} + \mathit{Bool} \\ \mathit{Env} &= \mathit{Var} \rightarrow \mathit{Val} \end{aligned}$$

Semantics rules:

$$\frac{}{\rho \vdash n \Rightarrow n} \quad \frac{}{\rho \vdash x \Rightarrow \rho(x)}$$

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

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

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

$$\frac{\rho \vdash E_1 \Rightarrow v_1 \quad [x \mapsto v_1]\rho \vdash E_2 \Rightarrow v}{\rho \vdash \text{let } x = E_1 \text{ in } E_2 \Rightarrow v}$$

Proc = Let + Procedures

$P \rightarrow E$

$E \rightarrow n$

| x

| $E + E$

| $E - E$

| iszero E

| if E then E else E

| let $x = E$ in E

| read

| proc $x E$

| $E E$

Example

- `let f = proc (x) (x-11)`
`in (f (f 77))`

Example

- `let f = proc (x) (x-11)`
`in (f (f 77))`
- `((proc (f) (f (f 77))) (proc (x) (x-11)))`

Free/Bound Variables of Procedures

- An occurrence of the variable x is *bound* when it occurs with definitions in the body or a procedure whose formal parameter is x .
- Otherwise, the variable is *free*.
- Examples:
 - ▶ `proc (y) (x+y)`
 - ▶ `proc (x) (let y = 1 in x + y + z)`
 - ▶ `proc (x) (proc (y) (x+y))`
 - ▶ `let x = 1 in proc (y) (x+y)`
 - ▶ `let x = 1 in proc (y) (x+y+z)`

Static vs. Dynamic Scoping

What is the result of the program?

```
let x = 1
in let f = proc (y) (x+y)
    in let x = 2
        in let g = proc (y) (x+y)
            in (f 1) + (g 1)
```

Static vs. Dynamic Scoping

What is the result of the program?

```
let x = 1
in let f = proc (y) (x+y)
    in let x = 2
        in let g = proc (y) (x+y)
            in (f 1) + (g 1)
```

Two ways to determine free variables of procedures:

- In *static scoping* (*lexical scoping*), the procedure body is evaluated in the environment where the procedure is defined (i.e. procedure-creation environment).
- In *dynamic scoping*, the procedure body is evaluated in the environment where the procedure is called (i.e. calling environment)

Exercises

What is the result of the program?

- In static scoping:
 - In dynamic scoping:
- 1

```
let a = 3
  in let p = proc (z) a
      in let f = proc (x) (p 0)
          in let a = 5
              in (f 2)
```
 - 2

```
let a = 3
  in let p = proc (z) a
      in let f = proc (a) (p 0)
          in let a = 5
              in (f 2)
```

Why Static Scoping?

Most modern languages use static scoping. Why?

- Reasoning about programs is much simpler in static scoping.
- In static scoping, renaming bound variables by their lexical definitions does not change the semantics, which is unsafe in dynamic scoping.

```
let x = 1
in let f = proc (y) (x+y)
    in let x = 2
        in let g = proc (y) (x+y)
            in (f 1) + (g 1)
```

- In static scoping, names are resolved at compile-time.
- In dynamic scoping, names are resolved only at runtime.

Example

- OCaml (static scoping)

```
# let x = 1 in
let f = fun y -> x + y in
let x = 2 in
let g = fun y -> x + y in
(f 1) + (g 1)
- : int = 5
```

- Python (dynamic scoping)

```
>>> x = 1
>>> def f(y): return x + y
>>> x = 2
>>> def g(y): return x + y
>>> f(1) + g(1)
6
```

Semantics of Procedures: Static Scoping

- Domain:

$$\begin{aligned} \mathit{Val} &= \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} \\ \mathit{Procedure} &= \mathit{Var} \times \mathit{E} \times \mathit{Env} \\ \mathit{Env} &= \mathit{Var} \rightarrow \mathit{Val} \end{aligned}$$

The procedure value is called *closures*. The procedure is closed in its creation environment.

Semantics of Procedures: Static Scoping

- Domain:

$$\begin{aligned} \mathit{Val} &= \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} \\ \mathit{Procedure} &= \mathit{Var} \times \mathit{E} \times \mathit{Env} \\ \mathit{Env} &= \mathit{Var} \rightarrow \mathit{Val} \end{aligned}$$

The procedure value is called *closures*. The procedure is closed in its creation environment.

- Semantics rules:

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

Examples

$$\overline{\square \vdash (\text{proc } (x) (x)) \mathbf{1} \Rightarrow \mathbf{1}}$$

Examples

$$\boxed{\vdash} \quad \begin{array}{l} \text{let } x = 1 \\ \text{in let } f = \text{proc } (y) \text{ (x+y)} \\ \quad \text{in let } x = 2 \\ \quad \quad \text{in (f 3)} \end{array} \Rightarrow 4$$

Semantics of Procedures: Dynamic Scoping

- Domain:

$$\begin{aligned} \mathit{Val} &= \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} \\ \mathit{Procedure} &= \mathit{Var} \times \mathit{E} \\ \mathit{Env} &= \mathit{Var} \rightarrow \mathit{Val} \end{aligned}$$

- Semantics rules:

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

Examples

$\square \vdash$ `let x = 1`
`in let f = proc (y) (x+y)`
`in let x = 2`
`in (f 3)` \Rightarrow **5**

cf) Multiple Argument Procedures

- We can get the effect of multiple argument procedures by using procedures that return other procedures.
- ex) a function that takes two arguments and return their sum:

```
let f = proc (x) proc (y) (x+y)
in ((f 3) 4)
```

Adding Recursive Procedures

The current language does not support recursive procedures, e.g.,

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

for which evaluation gets stuck:

$$\frac{[f \mapsto (x, f x, [])] \vdash f \Rightarrow (x, f x, []) \quad \frac{[x \mapsto 1] \vdash f \Rightarrow? \quad [x \mapsto 1] \vdash x \Rightarrow 1}{[x \mapsto 1] \vdash f x \Rightarrow?}}{[f \mapsto (x, f x, [])] \vdash (f 1) \Rightarrow?}$$

Two solutions:

- go back to dynamic scoping :-)
- modify the language syntax and semantics for procedure :-)

Recursion is Not Special in Dynamic Scoping

With dynamic scoping, recursive procedures require no special mechanism.
Running the program

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

via dynamic scoping semantics

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

proceeds well:

$$\overline{[] \vdash \text{let } f = \text{proc } (x) (f x) \text{ in } (f 1) \Rightarrow}$$

Recursion is Not Special in Dynamic Scoping

With dynamic scoping, recursive procedures require no special mechanism.
Running the program

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

via dynamic scoping semantics

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

proceeds well:

$$\frac{\begin{array}{c} \vdots \\ \hline [f \mapsto (x, f x), x \mapsto 1] \vdash f x \Rightarrow \\ \hline [f \mapsto (x, f x), x \mapsto 1] \vdash f x \Rightarrow \\ \hline [f \mapsto (x, f x)] \vdash f 1 \Rightarrow \end{array}}{\hline [] \vdash \text{let } f = \text{proc } (x) (f x) \text{ in } (f 1) \Rightarrow}$$

Adding Recursive Procedures

$P \rightarrow E$
 $E \rightarrow n$
| x
| $E + E$
| $E - E$
| iszero E
| if E then E else E
| let $x = E$ in E
| read
| letrec $f(x) = E$ in E
| proc $x E$
| $E E$

Example

```
letrec double(x) =  
  if zero?(x) then 0 else ((double (x-1)) + 2)  
in (double 1)
```

Semantics of Recursive Procedures

- Domain:

$$\begin{aligned} \mathit{Val} &= \mathbb{Z} + \mathit{Bool} + \mathit{Procedure} + \mathit{RecProcedure} \\ \mathit{Procedure} &= \mathit{Var} \times \mathit{E} \times \mathit{Env} \\ \mathit{RecProcedure} &= \mathit{Var} \times \mathit{Var} \times \mathit{E} \times \mathit{Env} \\ \mathit{Env} &= \mathit{Var} \rightarrow \mathit{Val} \end{aligned}$$

- Semantics rules:

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

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

Example

$$\overline{[] \vdash \text{letrec } f(x) = f\ x \text{ in } f\ 1 \Rightarrow}$$

Example

$$\frac{\frac{[f \mapsto (f, x, f x, \square)] \vdash f \Rightarrow (f, x, f x, \square)}{[f \mapsto (f, x, f x, \square)] \vdash f \Rightarrow (f, x, f x, \square)} \quad \frac{\frac{\vdots}{[x \mapsto 1, f \mapsto (f, x, f x, \square)] \vdash f x \Rightarrow} { [x \mapsto 1, f \mapsto (f, x, f x, \square)] \vdash f x \Rightarrow}}{[f \mapsto (f, x, f x, \square)] \vdash f \Rightarrow (f, x, f x, \square)} \quad \frac{[f \mapsto (f, x, f x, \square)] \vdash f \Rightarrow (f, x, f x, \square)}{\square \vdash \text{letrec } f(x) = f x \text{ in } f \Rightarrow}}$$

Mutually Recursive Procedures

$P \rightarrow E$

$E \rightarrow n$

| x

| $E + E$

| $E - E$

| iszero E

| if E then E else E

| let $x = E$ in E

| read

| letrec $f(x) = E$ in E

| letrec $f(x_1) = E_1$ and $g(x_2) = E_2$ in E

| proc $x E$

| $E E$

Example

```
letrec
  even(x) = if iszero(x) then 1 else odd(x-1)
  odd(x)  = if iszero(x) then 0 else even(x-1)
in (odd 13)
```

Semantics of Recursive Procedures

To support mutually recursive procedures, we need to extend the domain and the semantics:

- Domain:

$$\begin{aligned} \mathit{Val} &= \dots + \mathit{MRecProcedure} \\ \mathit{MRecProcedure} &= \mathit{Var} \times \mathit{Var} \times \mathit{E} \times \mathit{Var} \times \mathit{Var} \times \mathit{E} \times \mathit{Env} \end{aligned}$$

- Semantics rules:

$$\frac{[f \mapsto (f, x, E_1, g, y, E_2, \rho), g \mapsto (g, y, E_2, f, x, E_1, \rho)]\rho \vdash E_3 \Rightarrow v}{\rho \vdash \text{letrec } f(x) = E_1 \text{ and } g(y) = E_2 \text{ in } E_3 \Rightarrow v}$$

$$\rho \vdash E_1 \Rightarrow (f, x, E_f, g, y, E_g, \rho') \quad \rho \vdash E_2 \Rightarrow v$$

$$\frac{[x \mapsto v, f \mapsto (f, x, E_f, g, y, E_g, \rho'), g \mapsto (g, y, E_g, f, x, E_f, \rho')]\rho' \vdash E_f \Rightarrow v'}{\rho \vdash E_1 E_2 \Rightarrow v'}$$

Summary: The Proc Language

A programming language with expressions and procedures:

Syntax

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

Summary

Semantics

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

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

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

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

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

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

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