Practice Examples

- Give a derivation that proves the following
  (where \(\bullet\) is the empty environment)
  \[
  \begin{align*}
  \bullet; & \text{ let } x = 5 \text{ in let } y = 7 \text{ in } x+y \Rightarrow 12 \\
  \bullet; & \text{ let } x = \text{ let } x = 5 \text{ in } x+2 \text{ in } x+2 \Rightarrow 9 \\
  \bullet; & \text{ let } f = \text{ fun } x \rightarrow x+5 \text{ in } f 7 \Rightarrow 12 \\
  \bullet; & \text{ let } y = 5 \text{ in let } f = \text{ fun } x \rightarrow x+y \text{ in let } y = 6 \text{ in } f 7 \Rightarrow 12
  \end{align*}
  \]

- Using the dynamic scoping version of the function application rule, prove
  \[
  \begin{align*}
  \bullet; & \text{ let } y = 5 \text{ in } f = \text{ fun } x \rightarrow x+y \text{ in let } y = 6 \text{ in } f 7 \Rightarrow 13
  \end{align*}
  \]
Notation, Operations on Environments

- • is the empty environment (undefined for all ids)
- x:v is the environment that maps x to v and is undefined for all other ids
- If A and A’ are environments then A, A’ is the environment defined as follows

\[
(A, A')(id) = \begin{cases} 
A'(id) & \text{if } A'(id) \text{ defined} \\
A(id) & \text{if } A'(id) \text{ undefined but } A(id) \text{ defined} \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

- Idea: A’ “overwrites” definitions in A
- For brevity, can write •, A as just A

 Rule for Identifiers

\[
A(x) = v
\]

\[
A; x \Rightarrow v
\]

- x is an identifier
- To determine its value v “look it up” in A!
SOS Rules: Built-in Functions

- For arithmetic, comparison operations, etc.

\[
\begin{align*}
A; E_1 & \Rightarrow v_1 & A; E_2 & \Rightarrow v_2 \\
A; E_1 \text{ op } E_2 & \Rightarrow v_1 \text{ op } v_2
\end{align*}
\]

- For ::

\[
\begin{align*}
A; E_1 & \Rightarrow v_1 & A; E_2 & \Rightarrow v_2 \\
A; E_1 :: E_2 & \Rightarrow \langle v_1, v_2 \rangle
\end{align*}
\]

- Rules are recursive
- :: is implemented using pairing
  - Type system guarantees result is list

Trees of Semantic Rules

- When we apply rules to an expression, we actually get a tree
  - Corresponds to the recursive evaluation procedure
    - For example: \((3 + 4) + 5\)

\[
\begin{align*}
\bullet; 3 & \Rightarrow 3 & \bullet; 4 & \Rightarrow 4 \\
\bullet; (3 + 4) & \Rightarrow 7 & \bullet; 5 & \Rightarrow 5 \\
\bullet; (3 + 4) + 5 & \Rightarrow 12
\end{align*}
\]
Rule for Let Binding

We evaluate the first expression, and then evaluate the second expression in an environment extended to include a binding for x.

\[
\begin{align*}
A; E_1 & \Rightarrow v_1 \\
A, x:v_1; E_2 & \Rightarrow v_2 \\
A; \text{let } x = E_1 \text{ in } E_2 & \Rightarrow v_2
\end{align*}
\]

Example: let x = 1 in x+2

\[
\begin{align*}
A(x) & = v \\
A; x & \Rightarrow v \\
\text{•,x:1}; x & \Rightarrow 1 \\
\text{•,x:1}; 2 & \Rightarrow 2 \\
\text{•}; 1 & \Rightarrow 1 \\
\text{•,x:1}; x+2 & \Rightarrow 3 \\
\text{•}; \text{let } x = 1 \text{ in } x+2 & \Rightarrow 3
\end{align*}
\]
Example: let $x = 5$ in let $y = 7$ in $x+y$

\[
\begin{align*}
\text{; } & x:5, y:7; x \Rightarrow 5 \quad \text{; } x:5, y:7; y \Rightarrow 7 \\
\text{; } & x:5; 7 \Rightarrow 7 \quad \text{; } x:5, y:7; x+y \Rightarrow 12 \\
\text{; } & 5 \Rightarrow 5 \quad \text{; } x:5; \text{ let } y = 7 \text{ in } x+y \Rightarrow 12 \\
\text{; } & \text{ let } x = 5 \text{ in } \text{ let } y = 7 \text{ in } x+y \Rightarrow 12
\end{align*}
\]

Example: let $x = \text{ let } x = 5 \text{ in } x+2 \text{ in } x+2$

\[
\begin{align*}
\text{; } & x:5; x \Rightarrow 5 \quad \text{; } x:5; 2 \Rightarrow 2 \\
\text{; } & 5 \Rightarrow 5 \quad \text{; } x:5; x+2 \Rightarrow 7 \\
\text{; } & \text{ let } x = 5 \text{ in } x+2 \Rightarrow 7 \quad \text{; } x:7; x+2 \Rightarrow 2 \\
\text{; } & \text{ let } x = \text{ let } x = 5 \text{ in } x+2 \text{ in } x+2 \Rightarrow 9
\end{align*}
\]
Rule for Function Definitions

\[
\begin{array}{|c|c|}
\hline
- & \hline
A; \text{fun } x \rightarrow E & \Rightarrow (A, \lambda x.E) \\
\hline
\end{array}
\]

- The expression evaluates to a closure
  - The id and body in the closure come from the expression
  - The environment is the one in effect when the expression is evaluated
- This will be used to implement static scope

Rule for Function Application

\[
\begin{array}{|c|c|}
\hline
A; E_1 \Rightarrow (A', \lambda x.E) & \\
A; E_2 \Rightarrow v_2 & \\
A', x:v_2; E \Rightarrow v & \\
\hline
A; E_1 E_2 \Rightarrow v & \\
\hline
\end{array}
\]

- 1\textsuperscript{st} hypothesis: \( E_1 \) evaluates to a closure
- 2\textsuperscript{nd} hypothesis: \( E_2 \) produces a value (call by value!)
- 3\textsuperscript{rd} hypothesis: Body \( E \) in modified closure environment produces a value
- This last value is the result of the application
Example: \((\text{fun } x \rightarrow x + 3) \ 4\)

\[
\begin{align*}
\bullet; x:4; x & \Rightarrow 4 \quad \bullet; x:4; 3 \Rightarrow 3 \\
\bullet; \text{fun } x \rightarrow x + 3 & \Rightarrow (\bullet, \lambda x. x + 3) \\
\bullet; 4 & \Rightarrow 4 \\
\bullet; x:4; x + 3 & \Rightarrow 7 \\
\bullet; (\text{fun } x \rightarrow x + 3) & 4 \Rightarrow 7
\end{align*}
\]

Example: \((\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y)) \ 3 \ 4\)

\[
\begin{align*}
\bullet; (\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y)) & \Rightarrow (\bullet, \lambda x. (\text{fun } y \rightarrow x + y)) \\
\bullet; 3 & \Rightarrow 3 \\
\bullet; x:3; (\text{fun } y \rightarrow x + y) & \Rightarrow (x:3, \lambda y. (x + y)) \\
\bullet; (\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y)) & 3 \Rightarrow (x:3, \lambda y. (x + y)) \\
\text{Let } <\text{previous}> & = (\text{fun } x \rightarrow (\text{fun } y \rightarrow x + y)) \ 3
\end{align*}
\]
Example (cont.)

\[
\begin{align*}
\text{•, x:3, y:4; x ⇒ 3} & \quad \text{•, x:3, y:4; y ⇒ 4} \\
\text{•; <previous> ⇒ (x:3, λy.(x + y))} & \\
\text{•; 4 ⇒ 4} & \\
\text{x:3, y:4; (x + y) ⇒ 7} & \\
\text{•; (<previous> 4) ⇒ 7}
\end{align*}
\]

Example: let f = fun x → x+5 in f 7

\[
\begin{align*}
\text{•, x:7; x ⇒ 7} & \quad \text{•, x:7; 5 ⇒ 5} \\
\text{•, f:(•, λx.x+5); f ⇒ (•, λx.x+5)} & \\
\text{•, f:(•, λx.x+5); 7 ⇒ 7} & \\
\text{•, x:7; x+5 ⇒ 12} & \\
\text{•; fun x → x+5 ⇒ (•, λx.x+5)} & \quad \text{•, f:(•, λx.x+5); f 7 ⇒ 12} \\
\text{•; let f = fun x → x+5 in f 7 ⇒ 12}
\end{align*}
\]
Example: let \( y = 5 \) in let \( f = \) fun \( x \to x+y \) in let \( y = 6 \) in \( f \) 7

Static scoping uses environment from closure:

\[
\begin{align*}
\text{y:} & \ 5, \ \text{x:} & \ 7; \ x \Rightarrow 7 \\
\text{y:} & \ 5, \ \text{x:} & \ 7; \ y \Rightarrow 5 \\
\text{y:} & \ 5, \ f: (\text{y:} & \ 5, \ \lambda x.x+y), \ \text{y:} & \ 6; \ f \Rightarrow (\text{y:} & \ 5, \ \lambda x.x+y) \\
\text{y:} & \ 5, \ f: (\text{y:} & \ 5, \ \lambda x.x+y), \ \text{y:} & \ 6; \ y \Rightarrow 7 \\
\text{y:} & \ 5, \ x: & \ 7; \ x+y \Rightarrow 12
\end{align*}
\]

\[
\begin{align*}
\text{y:} & \ 5, \ f: (\text{y:} & \ 5, \ \lambda x.x+y), \ \text{y:} & \ 6; \ f \ 7 \Rightarrow 12 \\
\text{y:} & \ 5, \ f: (\text{y:} & \ 5, \ \lambda x.x+y); \ 6 \Rightarrow 6 \\
\text{y:} & \ 5, \ f: (\text{y:} & \ 5, \ \lambda x.x+y); \ \text{let} \ y = 6 \ \text{in} \ f \ 7 \Rightarrow 12 \\
\text{y:} & \ 5; \ \text{fun} \ x \to x+y \Rightarrow (\text{y:} & \ 5, \ \lambda x.x+y)
\end{align*}
\]

\[
\begin{align*}
\text{•;} & \ 5 \Rightarrow 5 \ \text{y:} & \ 5; \ \text{let} \ f = \ \text{fun} \ x \to x+y \ \text{in} \ \text{…} \ \text{in} \ f \ 7 \Rightarrow 12
\end{align*}
\]

\[
\begin{align*}
\text{•;} & \ \text{let} \ y = 5 \ \text{in} \ \text{let} \ f = \ \text{fun} \ x \to x+y \ \text{in} \ \text{let} \ y = 6 \ \text{in} \ f \ 7 \Rightarrow 12
\end{align*}
\]

Dynamic Scoping

- The previous rule for functions implements static scoping, since it implements closures
- We could easily implement dynamic scoping

\[
\begin{array}{c|c}
A; E_1 \Rightarrow (A', \ \lambda x. E) \\
A; E_2 \Rightarrow v \\
A, \ x:v_2; E \Rightarrow v \\
A; E_1 E_2 \Rightarrow v \\
\end{array}
\]

- The only difference is to use the current environment \( A \), not the environment \( A' \)
  - Easy to see the origins of the dynamic scoping bug!
Example: let $y = 5$ in let $f =$
\[
\text{fun } x \rightarrow x+y \text{ in let } y = 6 \text{ in } f 7
\]

\[
\begin{array}{|c|c|}
\hline
\text{Dynamic} & \text{scoping uses current environment} \\
\hline
y:5, f:(y:5, \lambda x.x+y), y:6, x:7; x \Rightarrow 7 & y:5, f:(y:5, \lambda x.x+y), y:6, x:7; y \Rightarrow 6 \\
\hline
y:5, f:(y:5, \lambda x.x+y), y:6; f \Rightarrow (y:5, \lambda x.x+y) & y:5, f:(y:5, \lambda x.x+y), y:6; 7 \Rightarrow 7 \\
\hline
y:5, f:(y:5, \lambda x.x+y), y:6, x:7; x+y \Rightarrow 13 & y:5, f:(y:5, \lambda x.x+y), y:6; f 7 \Rightarrow 13 \\
\hline
y:5, f:(y:5, \lambda x.x+y); 6 \Rightarrow 6 & y:5, f:(y:5, \lambda x.x+y); \text{let } y = 6 \text{ in } f 7 \Rightarrow 13 \\
\hline
y:5; \text{fun } x \rightarrow x+y \Rightarrow (y:5, \lambda x.x+y) & y:5; \text{let } f = \text{fun } x \rightarrow x+y \text{ in } \ldots \text{ in } f 7 \Rightarrow 13 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{•; } 5 \Rightarrow 5 & \text{y:5; let } f = \text{fun } x \rightarrow x+y \text{ in } \ldots \text{ in } f 7 \Rightarrow 13 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{•; let } y = 5 \text{ in let } f = \text{fun } x \rightarrow x+y \text{ in let } y = 6 \text{ in } f 7 \Rightarrow 13 \\
\hline
\end{array}
\]