LFP = LC + λ + function application + variables

E ::= n | b | ℓ | E iop E | E cop E | if E then E else E
| !E | E := E | skip | E; E | while E do E
| x | λx.E | E E | let x = E in E

where
- x ∈ V, an infinite set of variables
- n ∈ Z (integers), b ∈ B (booleans), ℓ ∈ L (locations)
- iop ∈ (integer-valued binary operations)
- cop ∈ (boolean-valued binary comparisons)

We focus on the (λ-calculus + let) part of LFP.

Application via substitution and its problems

Call by name

\[ \downarrow_{\text{cbn}}: \frac{\langle E_1, s \rangle \downarrow \langle \lambda x. E_1', s' \rangle \langle E_1'[E_2/x], s' \rangle \downarrow \langle V, s'' \rangle}{\langle (E_1 E_2), s \rangle \downarrow \langle V, s'' \rangle} \]

Call by value

\[ \downarrow_{\text{cbv}}: \frac{\langle E_1, s \rangle \downarrow \langle \lambda x. E_1', s' \rangle \langle E_2, s' \rangle \downarrow \langle V_2, s'' \rangle \langle E_1'[V_2/x], s'' \rangle \downarrow \langle V, s''' \rangle}{\langle (E_1 E_2), s \rangle \downarrow \langle V, s''' \rangle} \]

- Call-by-name and call-by-value are defined above via substitution.
- Substitution is:
  - dandy for nailing down sensible meanings of application.
  - stinko for everyday implementations.

E.g., An implementation via substitution constantly needs to modify a program's source code.

Idea  In place of substituting a value v for a variable x, keep a dictionary of variables and their values and when you need the value of x, look it up.

References

An environment is just a table of variables and associated values.

Consider an expression $e = \text{if } z \text{ then } x \text{ else } y + 2$.
- With environment $\{ x \mapsto 3, y \mapsto 4, z \mapsto \text{tt} \}$, $e$ evaluates to 3.
- With environment $\{ x \mapsto 8, y \mapsto 5, z \mapsto \text{ff} \}$, $e$ evaluates to 7.
- Etc.

lookup($\rho$, $x$) returns the value (if any) of $x$ in environment $\rho$.

update($\rho$, $x$, $v$) returns a new environment $\rho[x \mapsto v]$
($\rho[x \mapsto v]$ is just like $\rho$ except $x$ has value $v$.)

Evaluating variable $x$ in environment $\rho \equiv \text{lookup}(\rho, x)$.

---

**Revising call-by-value big-step semantics, 1**

Definition

$\rho \vdash \langle e, s \rangle \downarrow_{V} \langle v', s' \rangle$ means that expression $e$ with environment $\rho$ and state $s$ evaluates to value $v$ and state $s'$.

\[
\begin{align*}
\text{Var:} & \quad \rho \vdash \langle x, s \rangle \downarrow_{V} \langle v, s \rangle \quad (v = \text{lookup}(\rho, x)) \\
\text{Let:} & \quad \rho \vdash \langle e_1, s \rangle \downarrow_{V} \langle v_1, s' \rangle \quad \rho[x \mapsto v_1] \vdash \langle e_2, s' \rangle \downarrow_{V} \langle v_2, s'' \rangle \\
& \quad \rho \vdash \langle \text{let } x = e_1 \text{ in } e_2, s \rangle \downarrow_{V} \langle v_2, s'' \rangle
\end{align*}
\]

Examples/Exercises: Let $\rho = \{ x \mapsto 7, y \mapsto 3 \}$.
- $\rho \vdash \langle x + y, s \rangle \downarrow_{V} ??$
- $\rho \vdash \langle \text{let } x = 1 \text{ in } x + y, s \rangle \downarrow_{V} ??$
- $\rho \vdash \langle \text{let } x = 1 \text{ in } \langle \text{let } z = 11 \text{ in } x + y + z, s \rangle \downarrow_{V} ??$

---

**Revising call-by-value big-step semantics, 2**

Preliminary versions of these rules:

\[
\begin{align*}
\rho \vdash \langle e_1, s \rangle \downarrow_{V} \langle \lambda x.e'_1, s' \rangle \\
\rho \vdash \langle e_2, s' \rangle \downarrow_{V} \langle v_2, s'' \rangle
\end{align*}
\]

\[
\begin{align*}
\text{App:} & \quad \rho[x \mapsto v_2] \vdash \langle e'_1, s'' \rangle \downarrow_{V} \langle v, s''' \rangle \\
\rho \vdash \langle e_1, e_2, s \rangle \downarrow_{V} \langle v, s''' \rangle & \quad \text{Fun:} \quad \rho \vdash \langle \lambda x.e, s \rangle \downarrow_{V} \langle \lambda x.e, s \rangle
\end{align*}
\]

Examples/Exercises: Let $\rho = \{ x \mapsto 7, y \mapsto 3 \}$.
- $\rho \vdash \langle \text{let } f = \lambda x. (x + y) \text{ in } (f \ 10), s \rangle \downarrow_{V} ??$
- $\rho \vdash \langle \text{let } f = \lambda x. (x + y) \text{ in } \langle \text{let } y = 100 \text{ in } (f \ 10) \rangle, s \rangle \downarrow_{V} ??$
Dynamic Scoping, 1

Re: \( \lambda \)-expressions, functions, procedures, etc.,
there are two sorts of environments you have to worry about:

1. The environment in force when the function was created.
2. The environment in force when the function is applied.

\[ \text{Dynamic-App: } \rho \vdash \langle e_1, s \rangle \Downarrow \langle \lambda x.e'_1, s' \rangle \]

\[ \rho \vdash \langle e_2, s' \rangle \Downarrow \langle v_2, s'' \rangle \]

\[ \rho \vdash \langle (e_1 \ e_2), s \rangle \Downarrow \langle v, s''' \rangle \]

Example: Let \( \rho = \{ x \mapsto 7, \ y \mapsto 3 \} \) and consider

\[ \rho \vdash \langle \text{let } f = \lambda x.x + y \]
\[ \text{ in let } g = \lambda y.f(y + 100) \]
\[ \text{ in } ((f\ 10) + (g\ 0)), s \rangle \Downarrow \square \]

What goes right under dynamic scoping?

\[ \text{let } f = \lambda n. \text{ if } n \leq 0 \text{ then } 1 \text{ else } n * (f\ (n - 1)) \]
\[ \text{ in } (f\ 3) \]

History

Discovered and formalized in early (~1960) Lisp implementations.

Dynamic Scoping, 2

\[ \rho \vdash \langle e_1, s \rangle \Downarrow \langle \lambda x.e'_1, s' \rangle \]

\[ \rho \vdash \langle e_2, s' \rangle \Downarrow \langle v_2, s'' \rangle \]

\[ \text{Dynamic-App: } \rho[x \mapsto v_2] \vdash \langle e'_1, s'' \rangle \Downarrow \langle v, s''' \rangle \]

Under dynamic scoping, when you apply a function in environment

\( (\lambda x.e'_1) \ e_2 \) in environment \( \rho \)

you evaluate \( e'_1 \) in environment \( \rho[x \mapsto v_2] \).

Question:

Is this a bug or a feature?

Lexical Scoping, 1

Re: \( \lambda \)-expressions, functions, procedures, etc.,
there are two sorts of environments you have to worry about:

1. The environment in force when the function is created.
2. The environment in force when the function is applied.

- In human language, statements need to be understood in context:
  
  \[ \text{Such a fact is probable, but undoubtedly false.} \]
  
  —Edward Gibbon in “Decline and Fall of the Roman Empire”

- When Gibbon was writing “probable” meant “well-recommended”.
- So in reading Gibbon we have to use a 1700’s English dictionary.
- We pull a similar trick for functions.
A closure, $e\rho$, is an expression $e$ with an environment $\rho$ such that $fv(e) \subseteq \text{domain}(\rho)$, i.e., all of $e$’s free variables are in $\rho$’s dictionary.

**Ideas:**
- A $\lambda$-expression evaluates to a closure.
- When we create a $\lambda$-expression, we “close” it with its definition-time environment.

**Lexical Scoping, 2**

**Definition**

Let $e' \rho$, is an expression $e$ with an environment $\rho$ such that $fv(e) \subseteq \text{domain}(\rho)$, i.e., all of $e$’s free variables are in $\rho$’s dictionary.

**Examples/Exercises:** Let $\rho = \{ x \mapsto 7, y \mapsto 3 \}$.
- $\rho \vdash \langle \text{let } f = \lambda x. (x + y) \text{ in } (f 10), s \rangle \Downarrow ??$
- $\rho \vdash \langle \text{let } f = \lambda x. (x + y) \text{ in } (\text{let } y = 100 \text{ in } (f 10)), s \rangle \Downarrow ??$
- $\rho \vdash \langle \text{let } f = \lambda n. \text{ if } n \leq 0 \text{ then } 1 \text{ else } n \ast (f \ (n - 1)) \text{ in } (f 3), s \rangle \Downarrow ??$

**Lexical Scoping, 3**

**Lexical-App:**

Let $\rho = \{ x \mapsto 7, y \mapsto 3 \}$.
- $\rho \vdash \{ x \mapsto 7, y \mapsto 3 \}$
- $\rho \vdash \{ x \mapsto 7, y \mapsto 3 \}$
- $\rho \vdash \{ x \mapsto 7, y \mapsto 3 \}$

**Examples/Exercises:** Let $\rho = \{ x \mapsto 7, y \mapsto 3 \}$.
- $\rho \vdash \langle \text{let } f = \lambda x. (x + y) \text{ in } (f 10), s \rangle \Downarrow ??$
- $\rho \vdash \langle \text{let } f = \lambda x. (x + y) \text{ in } (\text{let } y = 100 \text{ in } (f 10)), s \rangle \Downarrow ??$
- $\rho \vdash \langle \text{let } f = \lambda n. \text{ if } n \leq 0 \text{ then } 1 \text{ else } n \ast (f \ (n - 1)) \text{ in } (f 3), s \rangle \Downarrow ??$

**Puzzle 1**

- The $e\rho$ notation is an effort to keep things from getting even-more heavy-handed.
- An alternative would be something like: $\text{close}(e, \rho)$.

What the value of $e_1$ in environment $\rho_1$ under call-by-value with
- (a) lexical scoping?
- (b) dynamic scoping?

$\rho_1 = [a \mapsto 1, b \mapsto 2]$

$e_1 = \text{let } q = \lambda a.(a + b) \text{ in }$

$\text{let } a = 5 \ast b \text{ in }$

$\text{let } b = a \ast b \text{ in }$

$(q 100)$
Puzzle 1(a): Call-by-value, lexical scoping

\( \rho_1 = [a \mapsto 1, \ b \mapsto 2] \)

\( e_1 = \text{let } q = \lambda a.(a + b) \text{ in} \)
\( \text{let } a = 5 \times b \text{ in} \)
\( \text{let } b = a \times b \text{ in} \)
\( (q \ 100) \)

\( \rho_5: \quad a \mapsto 100 \rightarrow \rho_1 \quad (a + b) \)

value of \( e_1 \rho_1 \): 102

Puzzle 1(b): Call-by-value, dynamic scoping

\( \rho_1 = [a \mapsto 1, \ b \mapsto 2] \)

\( e_1 = \text{let } q = \lambda a.(a + b) \text{ in} \)
\( \text{let } a = 5 \times b \text{ in} \)
\( \text{let } b = a \times b \text{ in} \)
\( (q \ 100) \)

\( \rho_5: \quad a \mapsto 100 \rightarrow \rho_1 \quad (a + b) \)

value of \( e_1 \rho_1 \): 120

Puzzle 2

\( \rho_1 = [a \mapsto 1, \ b \mapsto 2] \)

\( e_2 = \text{let } p = \lambda a.(a + b) \text{ in} \)
\( \text{let } q = \lambda b.(a + (p \ b)) \text{ in} \)
\( \text{let } a = 10 \text{ in} \)
\( \text{let } b = 20 \text{ in} \)
\( (q \ 100) \)

What is the value of \( e_2 \) in environment \( \rho_1 \) under call-by-value with
(a) lexical scoping?
(b) dynamic scoping?

Puzzle 2(a): Call-by-value, lexical scoping

\( \rho_1 = [a \mapsto 1, \ b \mapsto 2] \)

\( e_2 = \text{let } p = \lambda a.(a + b) \text{ in} \)
\( \text{let } q = \lambda b.(a + (p \ b)) \text{ in} \)
\( \text{let } a = 10 \text{ in} \)
\( \text{let } b = 20 \text{ in} \)
\( (q \ 100) \)

value of \( e_2 \rho_1 \): 103
Warning: The following is tormented LFP; return is as in HW10.

\begin{align*}
let\ mbox = \lambda x.\{(\text{let } bx = (new x) \text{ in } (\lambda y.\{ bx := !bx + y; \text{ return } !bx \})); \\
u = (mbox 10); \\
v = (mbox (100 + (u 5)))
\end{align*}

In more familiar notation, \textit{mbox} is roughly:

\begin{verbatim}
function mbox(x) = { 
    var bx = (new x); 
    return (function fow(v) 
    { bx := !bx + v; \text{ return } !bx }); 
}
\end{verbatim}

In Java terms: 
- box is a class 
- \textit{mbox} is a box-constructor 
- \textit{u} and \textit{v} are instance methods 
- \textit{bx} is an instance variable.

value of \(e_2\rho_1\): \(10 + (100 + 100) = 210\)
The Call-By-Name Version

Lexical Scoping, 6

The Call-By-Name Version

Puzzle 3(b): Call-by-name

Consider $\rho_0 \vdash (e_0, s_0) \triangleright (v_1, s_1)$.

What are $v_1$ and $s_1$ we use lexical scoping and

(a) call-by-value evaluation?

(b) call-by-name evaluation?

Puzzle 3

1. $\rho_0 = \emptyset$
2. $s_0 = [\ell \mapsto 0]$
3. $e_0 = \text{let}\,*\,g = \lambda x.\{ \ell : =!\ell + 1; \text{return } x \}$

\[ z = (g\,100) \]
\[ \text{in } (z + z + z) \]

What are $v_1$ and $s_1$ we use lexical scoping and

(a) call-by-value evaluation?

(b) call-by-name evaluation?
Puzzle 4

\( \rho_0 = \emptyset \)
\( s_0 = [\ell \mapsto 0] \)
\( e_0 = \text{let } g = \lambda x.\{ \ell : =! \ell + 1; \text{ return } x \}; \)
\( h = \lambda y.2; \)
\( \text{in } (h \ g) \)

Consider \( \rho_0 \vdash (e_0, s_0) \downarrow \ (v_1, s_1) \).

What are \( v_1 \) and \( s_1 \) we use lexical scoping and 
(a) call-by-value evaluation?  
(b) call-by-name evaluation?

---

Puzzle 4(a): Call-by-value

\( \rho_0 = \emptyset \)
\( s_0 = [\ell \mapsto 0] \)
\( e_0 = \text{let } g = \lambda x.\{ \ell : =! \ell + 1; \text{ return } x \}; \)
\( h = \lambda y.2; \)
\( \text{in } (h \ g) \)

What are \( v_1 \) and \( s_1 \) in
\( \rho_0 \vdash (e_0, s_0) \downarrow \!_V \ (v_1, s_1) \)?

Recursion under lexical scoping, 1

Recall:

\[
E ::= \ldots \mid \text{rec } x.E
\]

Informally: “rec \( x.E \)” reads recursively define \( x \) to be \( E \).

The big-step operational semantics is given by:

\[
\begin{align*}
\text{unfolding}_{\text{subst}}: & \quad \langle E[(\text{rec } x.E)/x], s \rangle \downarrow \langle V, s' \rangle \\
\text{rec } x.E, s \downarrow \langle V, s' \rangle & \quad \text{unfolding}_{\text{subst}}:
\end{align*}
\]
The substitution-based version of unfold

\[\text{unfolding}_{\text{subst}}: \quad \langle E[(\text{rec } x.E)/x], s \rangle \downarrow \langle V, s' \rangle \]

\[\langle \text{rec } x.E, s \rangle \downarrow \langle V, s' \rangle \]

The environment-based version of unfold

\[\text{unfolding}_{\text{env}}: \quad \rho[x \mapsto \text{(rec } x.E)] \vdash \langle E, s \rangle \downarrow \langle V, s' \rangle \]

\[\rho \vdash \langle \text{rec } x.E, s \rangle \downarrow \langle V, s' \rangle \]

Try:

\[\vdash \langle \text{rec } z,(\text{if } !\ell > 0 \text{ then } (\ell := !\ell - 1; z) \text{ else skip})\rangle, \{ \ell \mapsto 2 \} \downarrow ?? \]