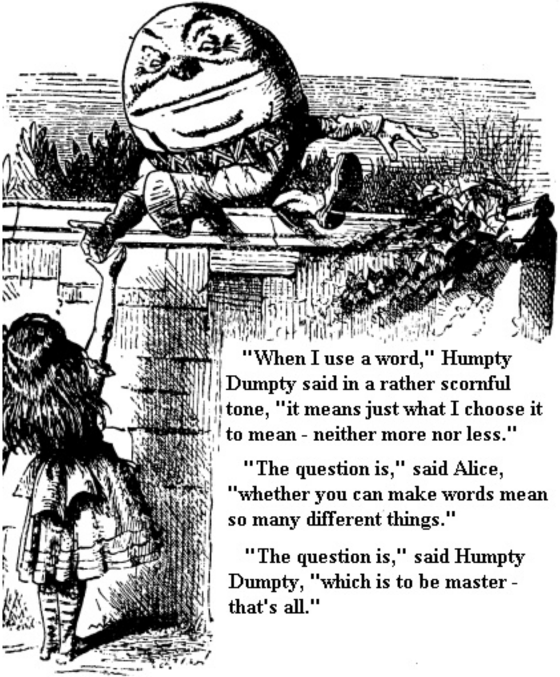
Names

A First Attempt

Jim Royer

CIS 352

February 26, 2019



References

- !!!! Slides for: *Language Semantics and Implementation Lectures 11 & 12*, by Richard Mayr and Colin Stirling, School of Informatics, University of Edinburgh, 2013. `http://www.inf.ed.ac.uk/teaching/courses/lsi/13Lsi11-12.pdf`, slide 9 onward
- Andrew Pitts' Lecture Notes on Semantics of Programming Languages: `http://www.inf.ed.ac.uk/teaching/courses/lsi/sempl.pdf`.
- *Semantics of programming languages Course Notes 2014-2015: Chapter 4, A simple functional language*, by Matthew Hennessy, Trinity College Dublin, University of Dublin, 2014. `https://www.scss.tcd.ie/Matthew.Hennessy/splexternal2015/LectureNotes/Notes14%20copy.pdf`
- William Cook, Anatomy of Programming Languages, Chapter 3, `http://www.cs.utexas.edu/~wcook/anatomy/anatomy.htm`

Building a better LC

It would be nice to have

- procedures & functions
- methods & classes & objects
- local variables
- constants
- …

To do this we need

- ways of naming things in LC
- ways of applying/invoking/calling/… the things named

Conventions:

- identifier $\equiv$ name $\equiv$ variable
- identifiers: $x, y, z, \ldots,$ but not ℓ

LLC = LC + let (an extended thought experiment)

$$P \ ::= \ \ldots \ | \ \textbf{let } x = P \textbf{ in } P'$$

P is a *phrase* (e.g., an expression or a command).

Note: "=" not ":=".

Example

- $\textbf{let } i = 42 \textbf{ in } i + 1$
- $\textbf{let } b = !\ell > 0 \textbf{ in if } b \textbf{ then } C_1 \textbf{ else } C_2$
- $\textbf{let } c = \{\, \ell := !\ell * !\ell';\ \ell' := \ell' - 1 \} \textbf{ in}$
  $\{\, \textbf{while } \ell' > 0 \textbf{ do } c \,\}$
- $\textbf{let } i = 42 \textbf{ in}$
  $\textbf{let } c_1 = \{\, \ell := !\ell + i \,\} \textbf{ in}$
  $\textbf{let } c_2 = \{\, \ell' := !\ell;\ c_1 \,\} \textbf{ in } C$

LLC = LC + let (an extended thought experiment)

$$P ::= \dots \mid \textbf{let } x = P \textbf{ in } P'$$

P is a *phrase* (e.g., an expression or a command).

Note: "=" not ":=".

Example

- $\textbf{let } i = 42 \textbf{ in } i + 1$
- $\textbf{let } b = !\ell > 0 \textbf{ in if } b \textbf{ then } C_1 \textbf{ else } C_2$
- $\textbf{let } c = \{\, \ell := !\ell * !\ell';\ \ell' := \ell' - 1 \} \textbf{ in }$
  $\{\, \textbf{while } \ell' > 0 \textbf{ do } c \,\}$
- $\textbf{let } i = 42 \textbf{ in }$
  $\textbf{let } c_1 = \{\, \ell := !\ell + i \,\} \textbf{ in }$
  $\textbf{let } c_2 = \{\, \ell' := !\ell;\ c_1 \,\} \textbf{ in } C$

- LLC is an extended thought experiment to start working out naming issues.
- We'll end up with something better.
- Note that in

$$\textbf{let } x = P \textbf{ in } P'$$

  P and P' can be arithmetic or boolean expressions or commands.
- Note that Scala, Swift, Rust, ... all use versions of "let".
- **N.B.** Many topics introduced here will be treated in more detail in LLC's successor.

LCC, continued

Example

- $\textbf{let } i = 42 \textbf{ in } i + 1$
- $\textbf{let } b = !\ell > 0 \textbf{ in if } b \textbf{ then } C_1 \textbf{ else } C_2$
- $\textbf{let } c = \{\, \ell := !\ell * !\ell';\ \ell' := \ell' - 1\} \textbf{ in}$
  $\{\, \textbf{while } \ell' > 0 \textbf{ do } c \,\}$
- $\textbf{let } i = 42 \textbf{ in}$
  $\textbf{let } c_1 = \{\, \ell := !\ell + i \,\} \textbf{ in}$
  $\textbf{let } c_2 = \{\, \ell' := !\ell;\ c_1 \,\} \textbf{ in } C$

In each of the above there are

- identifier definitions (e.g., $i = 42$)
- calls/references of the name inside of **in** (e.g., $i + 1$)

The semantics of let

Puzzle

Q: What is the value of $!\ell'$ after executing the following?

$$\ell := 0;\ \{\, \mathbf{let}\ x = !\ell + 1\ \mathbf{in}\ \{\, \ell := x;\ \ell' := x + 2\,\}\}$$

Q: In "**let** x = e **in** C", does the let substitute

(i) the expression e for x in C —*or*— **(ii)** the value of e for x in C?

An example that shows the difference between (i) and (ii):

$$\ell := 0;\ \{\, \mathbf{let}\ x = !\ell + 1\ \mathbf{in}\ \{\, \ell := x;\ \ell' := x + 2\,\}\}$$

$$\stackrel{?}{\equiv}\quad \ell := 0;\ \ell := !\ell + 1;\ \ell' := !\ell + 1 + 2$$

call-by-name

—*or*—

$$\stackrel{?}{\equiv}\quad \ell := 0;\ \ell'' := !\ell + 1;\ \ell := !\ell'';\ \ell' := !\ell'' + 2$$

call-by-value

LLC: Call-by-name and Call-by-value

Call-by-name

$$\frac{\langle P'[P/x], s \rangle \Downarrow_n \langle V', s' \rangle}{\langle \mathbf{let}\, x = P \,\mathbf{in}\, P', s \rangle \Downarrow_n \langle V', s' \rangle}$$

- $P'[P/x] \;\equiv\;$ the result of substituting P for x in P'
- *Defining substitution* ***correctly*** *is tricky, as we'll see.*

Call-by-value

$$\frac{\langle P, s \rangle \Downarrow_v \langle V, s_1 \rangle \qquad \langle P'[V/x], s_1 \rangle \Downarrow_v \langle V', s' \rangle}{\langle \mathbf{let}\, x = P \,\mathbf{in}\, P', s \rangle \Downarrow_v \langle V', s' \rangle}$$

- Good when P is an expression.
- Not-so-good when P is a command.
- Handy for handling parameters.

The next big goal:

Defining substitution (e.g., $P'[P/x]$) *precisely*!

The first subgoal:

Distinguishing between *free* and *bound* variables.

Variables, free and bound

$$\textbf{let } x \ = \ P \textbf{ in } P' \tag{1}$$

- Each (free) occurrence of x in P' is bound in (1) by the definition $x = P$ (called the ***defining (or binding) occurrence*** of x).
- A free occurrence of x in P' is in the ***scope*** this defining occurrence of x in (1).
- A free occurrence of x in P is ***NOT*** bound by this defining occurrence of x.

Consider:

1. $x + y$
2. $\textbf{let } x = y + x \textbf{ in } x + y$
3. $\textbf{let } y = 10 \textbf{ in } (\textbf{let } x = 20 \textbf{ in } x + (\textbf{let } x = y + x \textbf{ in } x + y))$

What is free? What is bound? and to what?

Variables, free and bound

What is free? What is bound? and to what?

1. $x^{\textit{free}} + y^{\textit{free}}$

2. $\textbf{let } x^{1} = y^{\textit{free}} + x^{\textit{free}} \textbf{ in } x^{1} + y^{\textit{free}}$

3. $$\begin{aligned} &\textbf{let } y^{1} = 10 \textbf{ in} \\ &\qquad (\textbf{let } x^{2} = 20 \textbf{ in} \\ &\qquad\qquad (x^{2} + (\textbf{let } x^{3} = y^{1} + x^{2} \textbf{ in } x^{3} + y^{1}))) \end{aligned}$$

Defining substitution (the easy/boring cases)

$$
\begin{aligned}
V[P/x] &= V \qquad\qquad (V \text{ is a value})\\
(E_1\ op\ E_2)[P/x] &= (E_1[P/x])\ op\ (E_2[P/x])\\
(\ell := E)[P/x] &= (\ell := (E[P/x])\\
(C_1;\ C_2)[P/x] &= (C_1[P/x]);\ (C_2[P/x])\\
(\mathbf{while}\ B\ \mathbf{do}\ C)[P/x] &= \mathbf{while}\ (B[P/x])\ \mathbf{do}\ (C[P/x])\\
(\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2)[P/x] &= \mathbf{if}\ (B[P/x])\ \mathbf{then}\ (C_1[P/x])\ \mathbf{else}\ (C_2[P/x])
\end{aligned}
$$

Defining substitution (the interesting cases)

$$y[P/x] = \begin{cases} P, & \text{if } x = y \\ y, & \text{if } x \neq y \end{cases}$$

$$(\textbf{let } y = P_1 \textbf{ in } P_2)[P/x] = \textbf{let } y = (P_1[P/x]) \textbf{ in } P_2$$

$$\text{(where } x = y\text{)}$$

$$(\textbf{let } y = P_1 \textbf{ in } P_2)[P/x] = \textbf{let } z = (P_1[P/x]) \textbf{ in } P_2[z/y][P/x]$$

$$\text{(where } (*)\text{)}$$

$(*)$ $x \neq y$ and $z \notin (\mathit{freeVars}(P) \cup \mathit{freeVars}(P_2) \cup \{ x \})$

Why so fussy in the last case? What could go wrong?

Capturing a variable (in C)

```
#define INCI(i) { int a=0; ++i; }
int main(void) {
    int a = 0, b = 0;
    INCI(a);
    INCI(b);
    printf("a is now %d, b is now %d", a, b);
    return 0;
}
```

Running the above through the C preprocessor produces:

```
int main(void) {
    int a = 0, b = 0;
    { int a=0; ++a; };
    { int a=0; ++b; };
    printf("a is now %d, b is now %d", a, b);
    return 0;
}
```

LCC with parameters

- $\textbf{let } c(x) = x + 3 \textbf{ in } c(8)$
- $\textbf{let } c(x, y) = x + y \textbf{ in } c(2, 3)$
- $\textbf{let } c(x, y) = (l := !l + y) \textbf{ in } c(2, 3)$
- $\textbf{let } add(x, y) = x + y$
  $\quad \textbf{in let } inc(x) = x + 1$
  $\qquad \textbf{in } add(4, inc(8))$
- $\textbf{let } c(x) = x + 3 \textbf{ in } c$
- $\textbf{let } c(x, p) = p(x * x)$
  $\quad \textbf{in let } inc(x) = x + 1$
  $\qquad \textbf{in } c(4, inc)$

Semantics

Call-by-name (without params)

$$\frac{\langle\, P'[P/x], s \,\rangle \Downarrow_n \langle\, V', s' \,\rangle}{\langle\, \mathbf{let}\, x = P \,\mathbf{in}\, P', s \,\rangle \Downarrow_n \langle\, V', s' \,\rangle}$$

Call-by-value (without params)

$$\frac{\langle\, P, s \,\rangle \Downarrow_v \langle\, V, s_1 \,\rangle \qquad \langle\, P'[V/x], s_1 \,\rangle \Downarrow_v \langle\, V', s' \,\rangle}{\langle\, \mathbf{let}\, x = P \,\mathbf{in}\, P', s \,\rangle \Downarrow_v \langle\, V', s' \,\rangle}$$

Puzzle: How to handle

$$\frac{?????????}{\langle\, \mathbf{let}\, c(x_1, \ldots, c_n) = P \,\mathbf{in}\, P', s \,\rangle \Downarrow_v \langle\, V', s' \,\rangle}$$

Semantics

Call-by-name (without params)

$$\frac{\langle P'[P/x], s \rangle \Downarrow_n \langle V', s' \rangle}{\langle \mathbf{let}\, x = P \,\mathbf{in}\, P', s \rangle \Downarrow_n \langle V', s' \rangle}$$

Call-by-value (without params)

$$\frac{\langle P, s \rangle \Downarrow_v \langle V, s_1 \rangle \qquad \langle P'[V/x], s_1 \rangle \Downarrow_v \langle V', s' \rangle}{\langle \mathbf{let}\, x = P \,\mathbf{in}\, P', s \rangle \Downarrow_v \langle V', s' \rangle}$$

Puzzle: How to handle

$$\frac{?????????}{\langle \mathbf{let}\, c(x_1, \ldots, c_n) = P \,\mathbf{in}\, P', s \rangle \Downarrow_v \langle V', s' \rangle}$$

- Formal vs. actual parameters
- cascading substitutions

Functions as “first class citizens”

- A function is another data-type.
  - We need a constructor for functions: λ
  - We need a way of using a function: application
- Instead of

$$\textbf{let } c(x) = P \textbf{ in } P'$$

  we write

$$\textbf{let } c = \lambda x.P \textbf{ in } P'$$

- So c names the function $\lambda x.P$
- In P' the function c may be applied to an argument: cM

Currying

- Instead of

$$\textbf{let } c(x_1, \cdots, x_n) = P \textbf{ in } P'$$

we write

$$\textbf{let } c = \lambda x_1. \cdots \lambda x_n.P \textbf{ in } P'$$

- So c names the function $\lambda x_1. \cdots \lambda x_n.P$
- In P' the function c may be applied to arguments: $cM_1 \ldots M_n$

[Stage direction: Make the directions of associativity clear!]

Example 1

$$\textbf{let } c(x) = x + 3 \textbf{ in } (c(8) + c(2))$$

becomes

$$\textbf{let } c = \lambda x.x + 3 \textbf{ in } (c\,8 + c\,2)$$

which is equivalent to

$$(\lambda x.x + 3)\,8 + (\lambda x.x + 3)\,2$$

but what does that mean??

Definition: $(\lambda x.P)P' \rightsquigarrow P[P'/x]$.

So

$$(\lambda x.x + 3)\,8 \rightsquigarrow (x + 3)[8/x] = 8 + 3$$
$$(\lambda x.x + 3)\,2 \rightsquigarrow (x + 3)[2/x] = 2 + 3$$

Example 2

$$\textbf{let } c(x,p) = p(x*x) \textbf{ in } (\textbf{let } inc(x) = x+1 \textbf{ in } c(4, inc))$$

$$\equiv$$

$$\textbf{let } c = \lambda x.\lambda p.p(x*x) \textbf{ in } (\textbf{let } inc = \lambda x.(x+1) \textbf{ in } c\,4\,inc)$$

$$\rightsquigarrow$$

$$\textbf{let } inc = \lambda x.(x+1) \textbf{ in } (\lambda x.\lambda p.p(x*x))\,4\,inc$$

$$\rightsquigarrow$$

$$(\lambda x.\lambda p.p(x*x))\,4\,(\lambda x.(x+1))$$

$$\rightsquigarrow$$

$$(\lambda p.p(4*4))\,(\lambda x.(x+1))$$

$$\rightsquigarrow$$

$$(\lambda x.(x+1))(4*4)$$

$$\rightsquigarrow$$

$$(4*4)+1$$

$$\rightsquigarrow$$

$$17$$

Defining let in terms of λ

$$\textbf{let } x = P \textbf{ in } P' \quad =_{\text{def}} \quad (\lambda x.P')P$$

- We'll use λ-expressions as primitive in our second attempt on names.