Scheme simplifies names

Simpler naming:
• Name stands for a mutable location
• Location holds values, updated with set
• Function is just another value
New Evaluation Judgment

Judgment $\langle e, \rho, \sigma \rangle \downarrow \langle v, \sigma' \rangle$

- Mappings in $\rho$ never change
- $\rho$ maps a name to a mutable location
- $\sigma$ is the store (contents of every location)

Intuition of the compiler writer:
- $\rho$ models the compiler’s “symbol table”
- $\sigma$ models the contents of registers and memory

Classic semantic technique
\(\mu\)Scheme vs Impcore

New abstract syntax:
- LET (keyword, names, expressions, body)
- LAMBDAX (formals, body)
- APPLY (exp, actuals)
Introduce local names into environment

(let ([x1 e1]
       ...
       [xn en])
  e)

Square brackets mean the same as round, but are easier to see
What McCarthy might have done

(let ([val x1 e1]
      ...
      [val xn en])
  e)

(But semantics of let, let*, letrec is much simpler)
Semantics of let binding

\[ \langle e_1, \rho, \sigma_0 \rangle \downarrow \langle v_1, \sigma_1 \rangle \]

\[ \langle e_2, \rho, \sigma_1 \rangle \downarrow \langle v_2, \sigma_2 \rangle \]

\[ \ell_1, \ell_2 \notin \text{dom } \sigma_2 \]

\[ \langle e, \rho\{x_1 \mapsto \ell_1, x_2 \mapsto \ell_2\}, \sigma_2\{\ell_1 \mapsto v_1, \ell_2 \mapsto v_2\} \rangle \downarrow \langle v, \sigma' \rangle \]

\[ \langle \text{LET}(\langle x_1, e_1, x_2, e_2 \rangle, e), \rho, \sigma \rangle \downarrow \langle v, \sigma' \rangle \] (LET2)
Function escapes!

-> (define to-the-n-minus-k (n k)
   (let
     ([x-to-the-n-minus-k (lambda (x)
                          (- (exp x n) k)])
      x-to-the-n-minus-k))
-> (val x-cubed-minus-27 (to-the-n-minus-k 3 27))
-> (x-cubed-minus-27 2)
-19
No need to name the escaping function

```
-> (define to-the-n-minus-k (n k)
   (lambda (x) (- (exp x n) k)))

-> (val x-cubed-minus-27 (to-the-n-minus-k 3 27))
-> (x-cubed-minus-27 2)
-19
```
The zero-finder

(define findzero-between (f lo hi)
  ; binary search
  (if (>= (+ lo 1) hi)
      hi
      (let ([mid (/ (+ lo hi) 2)])
        (if (< (f mid) 0)
            (findzero-between f mid hi)
            (findzero-between f lo mid))))
  
(define findzero (f) (findzero-between f 0 100))


Cube root of 27 and square root of 16

-> (findzero (to-the-n-minus-k 3 27))
3
-> (findzero (to-the-n-minus-k 2 16))
4
(define combine (p? q?)
    (lambda (x) (if (p? x) (q? x) #f)))

(define divvy (p? q?)
    (lambda (x) (if (p? x) #t (q? x))))

(val c-p-e (combine prime? even?))
(val d-p-o (divvy prime? odd?))

(c-p-e 9) == ?           (d-p-o 9) == ?
(c-p-e 8) == ?           (d-p-o 8) == ?
(c-p-e 7) == ?           (d-p-o 7) == ?
(define combine (p? q?)
  (lambda (x) (if (p? x) (q? x) #f)))

(define divvy (p? q?)
  (lambda (x) (if (p? x) #t (q? x)))))

(val c-p-e (combine prime? even?))
(val d-p-o (divvy prime? odd?))

(c-p-e 9) == #f (d-p-o 9) == #t
(c-p-e 8) == #f (d-p-o 8) == #f
(c-p-e 7) == #f (d-p-o 7) == #t
Algebraic laws when functions escape

Laws have nested applications on left-hand side:

- \(((\text{combine } p? \ q?)(x)) == (\text{if } (p? x) \ (q? x) \ #f)\)
- \(((\text{divvy } p? \ q?)(x)) == (\text{if } (p? x) \ #t \ (q? x))\)

One application for each define or lambda

Good place for syntactic sugar (short-circuit operators):

- \(((\text{combine } p? \ q?)(x)) == (\&\& (p? x) \ (q? x))\)
- \(((\text{divvy } p? \ q?)(x)) == (\|\| (p? x) \ (q? x))\)
Semantics of Lambda

Key Issue: Values of free variables

Static scoping:

Where \texttt{lambda} occurs, “look outward” for \( \rho \);
Capture that \( \rho \) for future reference.

\[
\langle \texttt{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle \langle \texttt{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho \rangle \rangle, \sigma \rangle
\]

Create closure in C implementation of \texttt{eval} by

case \texttt{LAMBDA}:
   
   return mkClosure(e->u.lambdax, env);
Applying Closures (Two Arguments)

Captured environment for free variables
Arguments for bound variables (≡ formal parameters)

\[
\langle e, \rho, \sigma \rangle \Downarrow \langle \text{LAMBDA}((x_1, x_2), e_c), \rho_c \rangle, \sigma_0 \rangle
\]

\[
\langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle
\]

\[
\langle e_2, \rho, \sigma_1 \rangle \Downarrow \langle v_2, \sigma_2 \rangle
\]

\[
\ell_1, \ell_2 \notin \text{dom} \sigma_2
\]

\[
\langle e_c, \rho_c \{x_1 \mapsto \ell_1, x_2 \mapsto \ell_2\}, \sigma_2 \{\ell_1 \mapsto v_1, \ell_2 \mapsto v_2\} \rangle \Downarrow \langle v, \sigma' \rangle
\]

\[
\langle \text{APPLY}(e, e_1, e_2), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle
\]

\text{APPLYCLOSURE2}

xs = f.u.closure.lambda.formals;
return eval(f.u.closure.lambda.body,
    bindalloclist(xs, vs, f.u.closure.env));