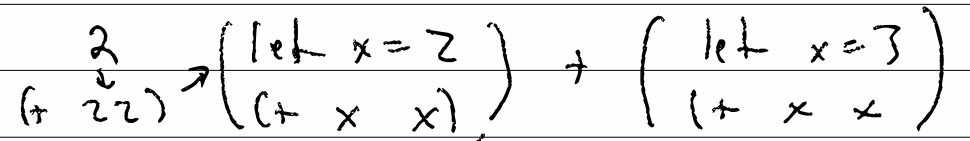
16-1/ Macro system

Our language is minimal and we implement things in the desugarer if possible.

Threads were just in the stdlib.
... could have been made by user



let double x = x + x
double 2 + double 6

16-2) desugar ["let", x, ":=", xe, "in", b]
⇒ ((λ (x) b) xe)

((λ (x) (+ x x)) 2) rather let x=2 in
x + x

(define (let x xe b)
((λ (x) b) xe))

(let x 2 (+ x x)) ← not a program (free vars)
↳ unbound ↳ unbound

functions cannot introduce binding forms

16-3/ (extend-the-desugarer!

```
(let x xe b)
((λ (x) b) xe))
```

C textual subst

```
#define MAX(x,y) x>y ? x : y
```

MAX(3,4) ⇒ 3>4 ? 3 : 4

```
#define TEMP(x) auto tmp = x
```

```
TEMP(3);
return tmp+2;
```

⇒

```
auto tmp = 3;
return tmp + 2;
```

11-4/ #define LET(x, xe) auto x = xe

```
LET(x, 3);          auto x=3;
LET(y, 4);    ⇒     auto y=4;
return x+y;         return x+y;
```

2* MAX(1+2, 3) → (2*1)+(2 > 3) ? (1+2) : 3)

#define MAX(x,y) ((x) > (y) ? (x) : (y))

MAX(i++, 3) → ((i++) > (3) ? (i++) : (3))

#define MAX(x,y) { auto tmpx = (x); auto tmpy = (y);
return x > y ? x : y; }

16-5/ Lisp / Scheme / Racket macros solve these problems.

define-syntax-rules

```
(define-syntax-rules
 [(let ([x xe] ...) be)
  ((λ (x ...) be) xe ...)])

(let ([x 2] [y 3]) (* x y))
 => ((λ (x y) (* x y)) 2 3)
```

16-6

```
(define-syntax-rules
  [(or)  false]
  [(or x y ...)
   (let ([tmp x])
     (if tmp tmp (or y ...)))])
```

template

pattern

```
(define-syntax-rules
  [(and)  true]
  [(and x)  x]
  [(and x y ...)
   (if x (and y ...) false)])
```

pattern →

case / clauses →

macro name

16-7/ define-syntax-rules : id × List(pair(pattern, template))

letm := "let", [< (let ([x xe] ...) be), ((λ (x ...) be) xe ...) >]

desugar will track a database of macros
ie a mapping from id to
List(pair (pattern, template))

desugar M [m, more ...] where m ∈ M :=
expand M[m] [m, more ...]

16-8/

pattern ↓

expand [< (let ([x xe] ...) be), ← def

template → ((λ (x ...) be) xe ...)>]

(let ([z 2]) (+ z z)) ← use

= ((λ (z) (+ z z)) 2)

match : pattern × sexpr → maybe (env)

env : id → sexpr

transcribe : template × env → sexpr

16-9/

pattern ↓

expand [< (let ([x xe] ...) be), ← def
template → ((λ (x ...) be) xe ...) >]

(let ([z 2]) (+ z z)) ← use

= ((λ (z) (+ z z)) 2)

match (let ([x xe] ...) be)
(let ([z 2]) (+ z z))

= [x ↦ (z), xe ↦ (2), be ↦ (+ z z)]

transcribe ((λ (x ...) be) xe ...)

= ((λ (z) (+ z z)) 2)

m '() '() = ∅

16-10/ m (cons pa pd) (cons ua ud) =
m pa ua ⊎ m pd ud

m (var x) u = [x ↦ u]

m (const n) n = ∅

m (list pat ...) u = merge (map (m pat) u)

tr '() env = '()

tr (cons ta td) env = cons (tr ta env)
(tr td env)

tr (var x) env = env(x)

tr (const n) env = (const n)

tr (list tmp ...) env = map (tr tmp) (unmerge

16-11 (define-syntax-rules

Matthew Flatt | Hygeni Macro

capture !

timestamps

colors

sets of scopes

```
[(or)  false]
[(or x) x]
[(or x y ...)
 (let ([tmp_1 x])
   (if tmp_1 tmp_1
       (or y ...)))]
```

```
(let ([tmp_0 4])
  (or false tmp_0))
```

⇓

4

```
(let ([tmp_0 4])
  (let ([tmp_1 false])
    (if tmp_1 tmp_1 tmp_0)))
```

⇒ false