22)

macro defn use

macro-expand : id × L(Pat × Templ) × Stx
⇒ Stx

exp ⇒

for pt in defn
(pat, template) := pt
(rho) p = patter-match pat use
when p
return (transcribe p templte)

match : pat × stx → env (id → stx)
match '() '() = ∅
match (Atomic a) a = ∅
same → num, string, bool, #:kw
match (Var v) use = [v ↦ use]
match (cons pa pd) (cons ua ud) =
⊕ (match pa ua)
(match pd ud)
match _ _ = fail

22-2 | ρ

transcribe : env(id ⇒ stx) × template ⇒ stx

tr ρ '() = '()

tr ρ (Atom a) = a

tr ρ (cons a d) = (cons (tr ρ a)
(tr ρ d))

tr ρ (var v) = if ρ(v) = (not false)
then return ρ(v)
o.w. fail

---

add ...s ρ : env(id ⇒ (nat, stx))

match (Var v) use = [v ↦ (0, use)]

tr ρ (var v) = (0, exp) = ρ(v)
exp

(dsr (mac x ...) (zero? x))

(mac 1 2 3) ⇒ (zero? (1 2 3))

(list ρ '...)

match (DDD p) (list? u) =

ps = map (match p) u

merge ps ---cut (match p —))

=

(λ (x) (match p x))

22-3/

```
tr p (DDD t) =
    map (tr _ t) (decompose p)

merge : (list pat-env) => pat-env
decompos : pat-env => (list patenv)
    merge^-1 = decompose

merge ps =
  if ps is mt => ∅
  if ps contains any fails => fail
  p0 = first ps
  for (v, b) in p0 where (lvl, _) = b
    add v ↦ (lvl+1,
                map (snd _ [v])
                    ps )
[x ↦ (0, a)] , [x ↦ (0, b)] =>
      [x ↦ (1, (a b))]
```

22-4/ decompose $[x \mapsto (1, (a\ b))] =$
$[x \mapsto (0, a)]$, $[x \mapsto (0, b)]$

decompose $[x \mapsto (0, a)$,
$y \mapsto (1, (b\ c))$
$z \mapsto (2, ((d\ e\ f)\ (g\ h\ \text{[illegible]})))]$

($sr (mac x (y z ...) ...)
xxxxxx) (list (x y z) ... ...)
(mac a (b d e f) (c g h [illegible]))
$\Rightarrow$ $[x \mapsto (0, a), b$
$y \mapsto (0,$ [illegible]$)$
$z \mapsto (1, (d\ e\ f))]$
$[x \mapsto (0, a)$
$y \mapsto (0, c)$
$z \mapsto (1, (g\ h))]$

22-5/ decompose p =

find vm in p s.t. level vm is largest

(lvl, uses) = p (vm)

len = length uses

build-list len

(λ i .

for (v, b) in p

(lvl, use) = b

if lvl = 0,

add (v, b) to ~~to~~

o.w.

add (v, (lvl-1, use[i]))

(dsms Or

[(_) Hf]

[(_ x) x]

[(_ x y ...)

(let ([v x])

(if v v (or y ...)))])

(or 1 2 3)

⇒ 1

(or Hf Hf 2)

⇒ 2

(let ([v 5]) (or Hf v))

(L [v 5] (L [v Hf] (if v v v)))