L4 grammar

```
    p ::= (e (l (x ...) e) ...)
    e ::= (let ((x e)) e)
        | (if e e e)

        | (e e ...)
        | (new-array e e)
        | (new-tuple e ...)
        | (aref e e)
        | (aset e e e)
        | (alen e)
        | (begin e e)
        | (print e)
        | (make-closure l e)
        | (closure-proc e)
        | (closure-vars e)
        | (biop e e)
        | (pred e)
        | num
        | x
        | l
 biop ::= + | - | * | cmpop
cmpop ::= < | <= | =
 pred ::= number? | a?
```

L4 vs. L3

```
p ::= (e (l (x ...) e) ...)
e ::= (let ((x e)) e)
   | (if e e e)

   | (e e ...)
   | (new-array e e)
   | (new-tuple e ...)
   | (aref e e)
   | (aset e e e)
   | (alen e)
   | (begin e e)
   | (print e)
   | (make-closure l e)
   | (closure-proc e)
   | (closure-vars e)
   | (biop e e)
   | (pred e)
   | num
   | x
   | l
biop ::= + | - | * | cmpop
cmpop ::= < | <= | =
pred ::= number? | a?
```

```
p ::= (e (l (x ...) e) ...)
e ::= (let ((x d)) e)
   | (if v e e)
   | d
d ::= (v v ...)
   | (new-array v v)
   | (new-tuple v ...)
   | (aref v v)
   | (aset v v v)
   | (alen v)

   | (print v)
   | (make-closure l v)
   | (closure-proc v)
   | (closure-vars v)
   | (biop v v)
   | (pred v)
   | v
v ::= x | l | num
biop ::= + | - | * | cmpop
cmpop ::= < | <= | =
pred ::= number? | a?
```

L4 → L3 Example

```
; results ≡ (new-tuple nat[passed] nat[tried])
; :update : int[bool] results → results
; called after each test case has run

>(:update 1 (new-tuple 3 4))
(new-tuple 4 5)
>(:update 0 (new-tuple 3 4))
(new-tuple 3 5)

(:update
 (pass? results)
 (new-tuple
  (let ((passed (aref 0 results)))
    (if pass? (+ 1 passed) passed))
  (+ 1 (aref 1 results))))
```

L4 → L3 Example

```
(:update
 (pass? results)
 (new-tuple
  (let ((passed (aref 0 results)))
    (if pass? (+ 1 passed) passed))
  (+ 1 (aref 1 results))))
```

L4 → L3 Example

```
(:update
 (pass? results)
 (new-tuple
  (let ((passed (aref 0 results)))
    (if pass? (+ 1 passed) passed))
  (+ 1 (aref 1 results))))
```

Next: lift `(aref 0 results)`

L4 → L3 Example

```
(:update
 (pass? results)
 (let ((x_1 (aref 0 results)))
   (new-tuple
    (let ((passed x_1))
      (if pass? (+ 1 passed) passed))
    (+ 1 (aref 1 results)))))
```

L4 → L3 Example

```
(:update
 (pass? results)
 (let ((x_1 (aref 0 results)))
   (new-tuple
    (let ((passed x_1))
      (if pass? (+ 1 passed) passed))
    (+ 1 (aref 1 results)))))
```

Next: lift `(let ((passed x_1)) ...)`

L4 → L3 Example

```
(:update
 (pass? results)
 (let ((x_1 (aref 0 results)))
   (let ((passed x_1))
     (new-tuple
      (if pass? (+ 1 passed) passed)
      (+ 1 (aref 1 results))))))
```

L4 → L3 Example

```
(:update
 (pass? results)
 (let ((x_1 (aref 0 results)))
   (let ((passed x_1))
     (new-tuple
      (if pass? (+ 1 passed) passed)
      (+ 1 (aref 1 results))))))
```

Next: lift `(if pass? ...)`

L4 → L3 Example

```
(:update
 (pass? results)
 (let ((x_1 (aref 0 results)))
   (let ((passed x_1))
     (if pass?
       (new-tuple
        (+ 1 passed)
        (+ 1 (aref 1 results)))
       (new-tuple
        passed
        (+ 1 (aref 1 results)))))))
```

L4 → L3 Example

```
(:update
 (pass? results)
 (let ((x_1 (aref 0 results)))
   (let ((passed x_1))
     (if pass?
       (new-tuple
        (+ 1 passed)
        (+ 1 (aref 1 results)))
       (new-tuple
        passed
        (+ 1 (aref 1 results)))))))
```

Next: lift `(+ 1 passed)`

L4 → L3 Example

```
(:update
 (pass? results)
 (let ((x_1 (aref 0 results)))
   (let ((passed x_1))
     (if pass?
       (let ((x_2 (+ 1 passed)))
         (new-tuple
          x_2
          (+ 1 (aref 1 results))))
       (new-tuple
        passed
        (+ 1 (aref 1 results)))))))
```

L4 → L3 Example

```
(:update
 (pass? results)
 (let ((x_1 (aref 0 results)))
   (let ((passed x_1))
     (if pass?
       (let ((x_2 (+ 1 passed)))
         (new-tuple
          x_2
          (+ 1 (aref 1 results))))
       (new-tuple
        passed
        (+ 1 (aref 1 results)))))))
```

Next: lift `(aref 1 results)`

L4 → L3 Example

```
(:update
 (pass? results)
 (let ((x_1 (aref 0 results)))
   (let ((passed x_1))
     (if pass?
       (let ((x_2 (+ 1 passed)))
         (let ((x_3 (aref 1 results)))
           (new-tuple x_2 (+ 1 x_3))))
       (new-tuple
        passed
        (+ 1 (aref 1 results)))))))
```

L4 → L3 Example

```
(:update
 (pass? results)
 (let ((x_1 (aref 0 results)))
   (let ((passed x_1))
     (if pass?
       (let ((x_2 (+ 1 passed)))
         (let ((x_3 (aref 1 results)))
           (new-tuple x_2 (+ 1 x_3))))
       (new-tuple
        passed
        (+ 1 (aref 1 results)))))))
```

Next: lift `(+ 1 x_3)`

L4 → L3 Example

```
(:update
 (pass? results)
 (let ((x_1 (aref 0 results)))
   (let ((passed x_1))
     (if pass?
       (let ((x_2 (+ 1 passed)))
         (let ((x_3 (aref 1 results)))
           (let ((x_4 (+ 1 x_3)))
             (new-tuple x_2 x_4))))
       (new-tuple
        passed
        (+ 1 (aref 1 results)))))))
```

L4 → L3 Example

```
(:update
 (pass? results)
 (let ((x_1 (aref 0 results)))
   (let ((passed x_1))
     (if pass?
       (let ((x_2 (+ 1 passed)))
         (let ((x_3 (aref 1 results)))
           (let ((x_4 (+ 1 x_3)))
             (new-tuple x_2 x_4))))
       (new-tuple
        passed
        (+ 1 (aref 1 results)))))))
```

Next: lift `(aref 1 results)`

L4 → L3 Example

```
(:update
 (pass? results)
 (let ((x_1 (aref 0 results)))
   (let ((passed x_1))
     (if pass?
       (let ((x_2 (+ 1 passed)))
         (let ((x_3 (aref 1 results)))
           (let ((x_4 (+ 1 x_3)))
             (new-tuple x_2 x_4))))
       (let ((x_5 (aref 1 results)))
         (new-tuple
          passed
           (+ 1 x_5))))))))
```

L4 → L3 Example

```
(:update
 (pass? results)
 (let ((x_1 (aref 0 results)))
   (let ((passed x_1))
     (if pass?
       (let ((x_2 (+ 1 passed)))
         (let ((x_3 (aref 1 results)))
           (let ((x_4 (+ 1 x_3)))
             (new-tuple x_2 x_4))))
       (let ((x_5 (aref 1 results)))
         (new-tuple
          passed
          (+ 1 x_5)))))))
```

Next: lift `(+ 1 x_5)`

L4 → L3 Example

```
(:update
 (pass? results)
 (let ((x_1 (aref 0 results)))
   (let ((passed x_1))
     (if pass?
       (let ((x_2 (+ 1 passed)))
         (let ((x_3 (aref 1 results)))
           (let ((x_4 (+ 1 x_3)))
             (new-tuple x_2 x_4))))
       (let ((x_5 (aref 1 results)))
         (let ((x_6 (+ 1 x_5)))
           (new-tuple passed x_6)))))))
```

L4 → L3 Example

```
(:update
 (pass? results)
 (let ((x_1 (aref 0 results)))
   (let ((passed x_1))
     (if pass?
       (let ((x_2 (+ 1 passed)))
         (let ((x_3 (aref 1 results)))
           (let ((x_4 (+ 1 x_3)))
             (new-tuple x_2 x_4))))
       (let ((x_5 (aref 1 results)))
         (let ((x_6 (+ 1 x_5)))
           (new-tuple passed x_6)))))))
```

Done

Normalization Rules: lift-app

If a **`d`** that's not already a **`v`** would be evaluated next, lift it into a **`let`** (if it's not already there).

```
(new-tuple
 (+ 1 passed)
 (+ 1 (aref 1 results)))

= (let ((x_1 (+ 1 passed)))
    (new-tuple
     x_1
     (+ 1 (aref 1 results))))
```

Normalization Rules: lift-if

If an **`if`** would be evaluated next, lift it and push the context into its branches.

```
    (new-tuple
     (if pass? (+ 1 passed) passed)
     (+ 1 (aref 1 results)))

 =  (if pass?
       (new-tuple
        (+ 1 passed)
        (+ 1 (aref 1 results)))
       (new-tuple
       passed
        (+ 1 (aref 1 results))))
```

Normalization Rules: lift-let

If a **`let`** would be evaluated next, lift it and push the context into its body.

```
(let ((x_1 (aref 0 results)))
  (new-tuple
   (let ((passed x_1))
     (if pass? (+ 1 passed) passed))
   (+ 1 (aref 1 results))))

= (let ((x_1 (aref 0 results)))
    (let ((passed x_1))
      (new-tuple
       (if pass? (+ 1 passed) passed)
       (+ 1 (aref 1 results)))))
```

Normalization Rules: lift-let

The **`let`**-bound variable must not be free in the context.

```
((let ((c a))
   (b c))
 c)

≠ (let ((c a))
    ((b c) c))
```

Rename the variable if necessary.

Normalization Rules: lift-let

We may consider a **`let`** to be “next” even if its RHS is a function or primitive application.

```
   (new-tuple
    (let ((passed (aref 0 results)))
      (if pass? (+ 1 passed) passed))
    (+ 1 (aref 1 results)))

 = (let ((passed (aref 0 results)))
     (new-tuple
      (if pass? (+ 1 passed) passed)
      (+ 1 (aref 1 results))))
```

Normalization Algorithm

Repeatedly applying the preceding rules turns an L4 expression into an L3 expression.

1. Following the order of evaluation, search the expression to find the first non-value that appears in a position where the L3 grammar expects a value.

2. Lift that non-value out of its context and into a `let`.

3. Repeat until we have an L3 program.

Normalization Algorithm

Revisit the sequence of intermediate expressions produced while converting **`:update`** to L3.

Normalization Algorithm

Revisit the sequence of intermediate expressions produced while converting **`:update`** to L3.

Observe: the portion of the expression above the most recently lifted expression is always already in L3. But the next step always begins by rescanning that region!

Normalization Algorithm

This observation suggests an optimization:

1. Following the order of evaluation, search the expression to find the first non-value that appears in a position where the L3 grammar expects a value.
2. Lift that non-value out of its context and into a **`let`**.
3. *Resume search from the point of the lifted non-value.*

Normalization Algorithm

For example, find the first expression to lift.

```
(f1 (f2 (f x) (g1 (g2 (g y)))))
```

Lift it into a **`let`**.

```
(let ([t (f x)])
  (f1 (f2 t (g1 (g2 (g y))))))
```

Resume with the next lift-candidate.

```
(let ([t (f x)])
  (f1 (f2 t (g1 (g2 (g y))))))
```

Normalization Algorithm

In general, the optimized algorithm alternates between two modes.

- *down mode* — find the next expression evaluated and go to up mode
- *up mode* — lift the expression into a **`let`**, if necessary, and proceed in down mode on the next expression evaluated.

Normalization Algorithm

Let's step through some examples of the optimized algorithm. We'll illustrate its progress by annotating the expression it rewrites.

- "⬇" marks the current position when in down mode.
- "⬆" marks the current position when in up mode.
- "●" marks where to place the next lifted expression.

Algorithm Example: nested application

```
(a ((b c) ((d e) f)))
```

Algorithm Example: nested application

```
(a ((b c) ((d e) f)))
```

Algorithm Example: nested application

```
(a ((b c) ((d e) f)))
```

Algorithm Example: nested application

```
(a ((b c) ((d e) f)))
```

Algorithm Example: nested application

```
(a ((b c) ((d e) f)))
```

Algorithm Example: nested application

```
(a ((b c) ((d e) f)))
```

Algorithm Example: nested application

```
(a ((b c) ((d e) f)))
```

Algorithm Example: nested application

```
(a ((b c) ((d e) f)))
```

Algorithm Example: nested application

```
(a ((b c) ((d e) f)))
```

Algorithm Example: nested application

```
(a ((b c) ((d e) f)))
```

Algorithm Example: nested application

```
(let ((x_1 (b c)))
  (a (x_1 ((d e) f))))
```

Algorithm Example: nested application

```
(let ((x_1 (b c)))
  (a (x_1 ((d e) f))))
```

Algorithm Example: nested application

```
(let ((x_1 (b c)))
  (a (x_1 ((d e) f))))
```

Algorithm Example: nested application

```
(let ((x_1 (b c)))
  (a (x_1 ((d e) f))))
```

Algorithm Example: nested application

```
(let ((x_1 (b c)))
  (a (x_1 ((d e) f))))
```

Algorithm Example: nested application

```
(let ((x_1 (b c)))
  (a (x_1 ((d e) f))))
```

Algorithm Example: nested application

```
(let ((x_1 (b c)))
  (a (x_1 ((d e) f))))
```

Algorithm Example: nested application

```
(let ((x_1 (b c)))
  (let ((x_2 (d e)))
    (a (x_1 (x_2 f)))))
```

Algorithm Example: nested application

```
(let ((x_1 (b c)))
  (let ((x_2 (d e)))
    (a (x_1 (x_2 f)))))
```

Algorithm Example: nested application

```
(let ((x_1 (b c)))
  (let ((x_2 (d e)))
    (a (x_1 (x_2 f)))))
```

Algorithm Example: nested application

```
(let ((x_1 (b c)))
  (let ((x_2 (d e)))
    (let ((x_3 (x_2 f)))
      (a (x_1 x_3)))))
```

Algorithm Example: nested application

```
(let ((x_1 (b c)))
  (let ((x_2 (d e)))
    (let ((x_3 (x_2 f)))
      (let ((x_4 (x_1 x_3)))
        (a x_4)))))
```

Algorithm Example: nested application

```
(let ((x_1 (b c)))
  (let ((x_2 (d e)))
    (let ((x_3 (x_2 f)))
      (let ((x_4 (x_1 x_3)))
        (a x_4)))))
```

Algorithm Example: nested let

```
((let ((x (a b))) (c x))
 (d (let ((y e)) (f y))))
```

Algorithm Example: nested let

```
((let ((x (a b))) (c x))
 (d (let ((y e)) (f y))))
```

Algorithm Example: nested let

```
((let ((x (a b))) (c x))
 (d (let ((y e)) (f y))))
```

Algorithm Example: nested let

```
((let ((x (a b))) (c x))
 (d (let ((y e)) (f y))))
```

Algorithm Example: nested let

```
((let ((x (a b))) (c x))
 (d (let ((y e)) (f y))))
```

Algorithm Example: nested let

```
((let ((x (a b))) (c x))
 (d (let ((y e)) (f y))))
```

Algorithm Example: nested let

```
((let ((x (a b))) (c x))
 (d (let ((y e)) (f y))))
```

Algorithm Example: nested let

```
((let ((x (a b))) (c x))
 (d (let ((y e)) (f y))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  ((c x)
   (d (let ((y e)) (f y)))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  (((c x)
   (d (let ((y e)) (f y))))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  ((c x)
   (d (let ((y e)) (f y)))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  ((c x)
   (d (let ((y e)) (f y)))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  ((c x)
   (d (let ((y e)) (f y)))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  ((c x)
   (d (let ((y e)) (f y)))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  (let ((x_1 (c x)))
    (x_1
     (d
      (let ((y e))
        (f y)))))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  (let ((x_1 (c x)))
    (x_1
     (d
      (let ((y e))
        (f y)))))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  (let ((x_1 (c x)))
    (x_1
      (d
        (let ((y e))
          (f y))))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  (let ((x_1 (c x)))
    (x_1
     (d
      (let ((y e))
        (f y)))))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  (let ((x_1 (c x)))
    (x_1
     (d
      (let ((y e))
        (f y)))))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  (let ((x_1 (c x)))
    (x_1
     (d
      (let ((y e))
        (f y))))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  (let ((x_1 (c x)))
    (let ((y e))
      (x_1 (d (f y))))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  (let ((x_1 (c x)))
    (let ((y e))
      (x_1 (d (f y))))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  (let ((x_1 (c x)))
    (let ((y e))
      (x_1 (d (f y))))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  (let ((x_1 (c x)))
    (let ((y e))
      (x_1 (d (f y)))))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  (let ((x_1 (c x)))
    (let ((y e))
      (x_1 (d (f y))))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  (let ((x_1 (c x)))
    (let ((y e))
      (x_1 (d (f y))))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  (let ((x_1 (c x)))
    (let ((y e))
      (let ((x_2 (f y)))
        (x_1 (d x_2))))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  (let ((x_1 (c x)))
    (let ((y e))
      (let ((x_2 (f y)))
        (let ((x_3 (d x_2)))
          (x_1 x_3))))))
```

Algorithm Example: nested let

```
(let ((x (a b)))
  (let ((x_1 (c x)))
    (let ((y e))
      (let ((x_2 (f y)))
        (let ((x_3 (d x_2)))
          (x_1 x_3))))))
```

Algorithm Example: nested if

```
(((if (a b)
     c
     (if d e f))
  g)
 h)
```

Algorithm Example: nested if

```
(((if (a b)
      c
      (if d e f))
   g)
  h)
```

Algorithm Example: nested if

```
(((if (a b)
     c
     (if d e f))
  g)
 h)
```

Algorithm Example: nested if

```
(((if (a b)
    c
    (if d e f))
  g)
 h)
```

Algorithm Example: nested if

```
(((if (a b)
    c
    (if d e f))
  g)
 h)
```

Algorithm Example: nested if

```
(((if (a b)
    c
    (if d e f))
  g)
 h)
```

Algorithm Example: nested if

```
(((if (a b)
      c
      (if d e f))
   g)
  h)
```

Algorithm Example: nested if

```
(((if (a b)
    c
    (if d e f))
  g)
 h)
```

Algorithm Example: nested if

```
(((if (a b)
    c
    (if d e f))
  g)
 h)
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (((c g) h)
    (((if d e f) g)
     h)))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    ((c g) h)
    (((if d e f) g)
     h)))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    ((c g) h)
    (((if d e f) g)
     h)))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    ((c g) h)
    (((if d e f) g)
     h)))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    ((c g) h)
    (((if d e f) g)
     h)))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (((if d e f) g)
     h)))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (((if d e f) g)
     h)))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (((if d e f) g)
     h)))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (((if d e f) g)
     h)))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (((if d e f) g)
     h)))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (((if d e f) g)
     h)))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (if d
      ((e g) h)
      ((f g) h))))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (if d
      ((e g) h)
      ((f g) h))))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (if d
      ((e g) h)
      ((f g) h))))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (if d
      ((e g) h)
      ((f g) h))))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (if d
      ((e g) h)
      ((f g) h))))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (if d
      (let ((x_3
               (e g)))
        (x_3 h))
      ((f g) h))))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (if d
      (let ((x_3
               (e g)))
        (x_3 h))
      ((f g) h))))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (if d
      (let ((x_3
               (e g)))
        (x_3 h))
      ((f g) h))))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (if d
      (let ((x_3
                (e g)))
        (x_3 h))
      ((f g) h))))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (if d
      (let ((x_3
              (e g)))
        (x_3 h))
      ((f g) h))))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (if d
      (let ((x_3
                (e g)))
        (x_3 h))
      ((f g) h))))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (if d
      (let ((x_3
               (e g)))
        (x_3 h))
      ((f g) h))))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (if d
      (let ((x_3
                 (e g)))
        (x_3 h))
      ((f g) h))))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (if d
      (let ((x_3
              (e g)))
        (x_3 h))
      (let ((x_4
              (f g)))
        (x_4 h))))))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (if d
      (let ((x_3
              (e g)))
        (x_3 h))
      (let ((x_4
              (f g)))
        (x_4 h)))))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (if d
      (let ((x_3
                (e g)))
        (x_3 h))
      (let ((x_4
                (f g)))
        (x_4 h))))))
```

Algorithm Example: nested if

```
(let ((x_1 (a b)))
  (if x_1
    (let ((x_2 (c g)))
      (x_2 h))
    (if d
      (let ((x_3
              (e g)))
        (x_3 h))
      (let ((x_4
              (f g)))
        (x_4 h)))))
```

Algorithm Implementation

We'll implement the algorithm for a subset of L4.

```
e ::= (e e)
    | (let ((x e)) e)
    | (if e e e)
    | v
v ::= x | number
```

Extending the implementation to the rest of L4 is assignment 4.

Algorithm Implementation

The algorithm's execution defines a sequence of partially normalized expressions, each of which can be divided into three pieces:

1. the expression at the arrow,
2. the portion of the expression outside the circle, and
3. the portion of the expression outside (1) but inside (2).

(2) is irrelevant in subsequent steps, since it's already fully normalized, but we need a representation for (3).

Algorithm Implementation

A `context` is an expression with a hole in it. We'll represent it inside-out, so we can easily access what's just outside the arrow.

```
(define-type context
  [let-ctxt (x var?)
            (b L4-e?)
            (k context?)]
  [if-ctxt (t L4-e?)
           (e L4-e?)
           (k context?)]
  [fun-ctxt (a L4-e?)
            (k context?)]
  [arg-ctxt (f val?)
            (k context?)]
  [no-ctxt])
```

Algorithm Implementation

For this partially normalized expression

```
(...
 ((let ((x (a (b c))))
    d)
  e))
```

we represent the unnormalized context outside the arrow with this structure

```
(arg-ctxt
 'a
 (let-ctxt
  'x 'd
  (fun-ctxt 'e (no-ctxt))))
```

Algorithm Implementation

A pair of mutually recursive functions implement the steps shown in the illustrated traces.

```
; find: L4-e context → L3-e
; find takes the next step when a
; downward arrow points to e. k
; records the context between the
; arrow and the enclosing circle
(define (find e k)
  ...)

; fill: L3-d context → L3-e
; fill does the same for an upward
; arrow
(define (fill d k)
  ...)
```

Algorithm Implementation

We'll start with **`find`**, which dispatches on the form of the expression at the arrow.

```
(define (find e k)
  (match e
    [`(,f ,a)
     ...]
    [`(let ([,x ,r]) ,b)
     ...]
    [`(if ,c ,t ,e)
     ...]
    [(? val?)
     ...]))
```

Algorithm Implementation

When the arrow points down at an application, the search proceeds into the function position, extending the context with a **`fun-ctxt`** layer.

```
(define (find e k)
  (match e
    ...
    [`(,f ,a)
     (find f (fun-ctxt a k))]
    ...))
```

Algorithm Implementation

When the arrow points down at a **`let`**, the search proceeds into the right-hand side, extending the context with a **`let-ctxt`** layer.

```
(define (find e k)
  (match e
    ...
    [`(let ([,x ,r]) ,b)
     (find r (let-ctxt x b k))]
    ...))
```

Algorithm Implementation

When the arrow points down at an **`if`**, the search proceeds into the test position, extending the context with an **`if-ctxt`** layer.

```
(define (find e k)
  (match e
    ...
    [`(if ,c ,t ,e)
     (find c (if-ctxt t e k))]
    ...))
```

Algorithm Implementation

When the arrow points down at a value, leave it in place by calling `fill` (which will resume the search for the next expression by examining the context).

```
(define (find e k)
  (match e
    ...
    [(? val?)
     (fill e k)]
    ...))
```

Algorithm Implementation

We'll now define `fill`, which dispatches on the form of the enclosing context.

```
(define (fill d k)
  (type-case context k
    [let-ctxt (x b k)
              ...]
    [if-ctxt (t e k)
             ...]
    [fun-ctxt (a k)
              ...]
    [arg-ctxt (f k)
              ...]
    [no-ctxt ()
             ...]))
```

Algorithm Implementation

In defining `fill`'s cases, it will be helpful to examine some examples. For example, consider these steps, in which the context is a `fun-ctxt`.

```
  (let ((x_1 (b c)))
    (a (x_1 ((d e) f))))

→ (let ((x_1 (b c)))
    (a (x_1 ((d e) f))))
```

```
  (let ((x_1 (b c)))
    (a (x_1 ((d e) f))))

→ (let ((x_1 (b c)))
    (let ((x_2 (d e)))
      (a (x_1 (x_2 f)))))
```

Algorithm Implementation

These examples suggest that the `fun-ctxt` case has two sub-cases:

- when the function position is a value, leave it in place and continue with the argument position;
- otherwise, lift it into a `let` then do the same.

Algorithm Implementation

The second sub-case lifts **d** just outside of **k** making it the last **let** in the normalized portion of the program.

```
(define (fill d k)
  (type-case context k
    ...
    [fun-ctxt
     (a k)
     (if (val? d)
         (find a (arg-ctxt d k))
         (let ([x (fresh-var)])
           `(let ([,x ,d])
              ,(find a
                     (arg-ctxt x k)))))]
    ...))
```

Algorithm Implementation

Now consider these examples, in which the context is an **`arg-ctxt`**.

```
  (let ((x_1 (b c)))
    (a (x_1 ((d e) f))))

→ (let ((x_1 (b c)))
    (a (x_1 ((d e) f))))
```

```
  (let ((x_1 (b c)))
    (let ((x_2 (d e)))
      (a (x_1 (x_2 f)))))

→ (let ((x_1 (b c)))
    (let ((x_2 (d e)))
      (let ((x_3 (x_2 f)))
        (a (x_1 x_3)))))
```

Algorithm Implementation

These examples suggest that the **`arg-ctxt`** case has two sub-cases:

- when the argument position is a value, rebuild the application and examine its enclosing context;
- otherwise, lift it into a **`let`** then do the same.

Algorithm Implementation

```
(define (fill d k)
  (type-case context k
    ...
    [arg-ctxt
     (f k)
     (if (val? d)
         (fill `(,f ,d) k)
         (let ([x (fresh-var)])
           `(let ([,x ,d])
              ,(fill `(,f ,x) k))))]
    ...))
```

Algorithm Implementation

Now consider these examples, in which the context is a **`let-ctxt`**.

```
 ((let ((x (a b))) (c x))
  (d (let ((y e)) (f y))))

→ (let ((x (a b)))
    ((c x)
     (d
      (let ((y e))
        (f y))))))
```

```
  (let ((x (a b)))
    (let ((x_1 (c x)))
      (x_1
       (d
        (let ((y e))
          (f y)))))))

→ (let ((x (a b)))
    (let ((x_1 (c x)))
      (let ((y e))
        (x_1 (d (f y))))))
```

Algorithm Implementation

These examples suggest a single **`let-ctxt`** case: lift the entire **`let`** (whether or not the right-hand side is a value), pushing the context into the body of the **`let`**.

Algorithm Implementation

```
(define (fill d k)
  (type-case context k
    ...
    [let-ctxt
     (x b k)
     `(let ([,x ,d])
        ,(find b k))]
    ...))
```

Algorithm Implementation

Now consider these examples, in which the context is an **if-ctxt**.

```
 (((if (a b) c (if d e f))
   g)
  h)

→ (let ((x_1 (a b)))
    (if x_1
      ((c g) h)
      (((if d e f) g) h)))
```

```
 (let ((x_1 (a b)))
   (if x_1
     (let ((x_2 (c g)))
       (x_2 h))
     (((if d e f) g) h)))

→ (let ((x_1 (a b)))
   (if x_1
     (let ((x_2 (c g)))
       (x_2 h))
     (if d
       ((e g) h)
       ((f g) h))))
```

Algorithm Implementation

These examples suggest that the `if-ctxt` case has two sub-cases:

- if the test position is a value, leave it in place and push the context into the branches;
- otherwise, lift it into a `let` then do the same.

Algorithm Implementation

```
(define (fill d k)
  (type-case context k
    ...
    [if-ctxt
     (t e k)
     (if (val? d)
         `(if ,d
              ,(find t k)
              ,(find e k))
         (let ([x (fresh-var)])
           `(let ([,x ,d])
              (if ,x
                  ,(find t k)
                  ,(find e k)))))]
    ...))
```

Algorithm Implementation

There's one more case, **no-ctxt**. When there's nothing between the (normalized) expression at the (upward) arrow and the normalized portion of the program, we're done.

```
(define (fill d k)
  (type-case context k
    ...
    [no-ctxt () d]
    ...))
```

Algorithm Implementation

The top-level function **`norm`** calls **`find`** with the empty context.

```
; norm: L4-e → L3-e
(define (norm e)
  (find e (no-ctxt)))
```

Context Duplication

`if`-lifting rule duplicates the content. What happens when an **`if`** appears in the context of an **`if`**?

```
   (if (if x1 x2 x3) x4 x5)
 = (if x1 (if x2 x4 x5) (if x3 x4 x5))
```

`x4` and **`x5`** each appear twice.

Context Duplication

What about an **`if`** outside an **`if`** outside an **`if`**?

```
  (if (if (if x1 x2 x3) x4 x5) x6 x7)

= (if x1
    (if x2 (if x4 x6 x7) (if x5 x6 x7))
    (if x3 (if x4 x6 x7) (if x5 x6 x7)))
```

`x6` and **`x7`** each appear four times.

Context Duplication

What about an **`if`** outside an **`if`** outside an **`if`** outside an **`if`**?

```
 (if (if (if (if x1 x2 x3) x4 x5) x6 x7)
   x8
   x9)

= (if x1
    (if x2
      (if x4 (if x6 x8 x9) (if x7 x8 x9))
      (if x5 (if x6 x8 x9) (if x7 x8 x9)))
    (if x3
      (if x4 (if x6 x8 x9) (if x7 x8 x9))
      (if x5
        (if x6 x8 x9)
        (if x7 x8 x9))))
```

`x8` and **`x9`** each appear eight times.

Context Duplication

Ignore this problem in assignment 4.

Context Duplication

We can avoid exponential growth by turning the context into a function, to be tail-called by the branches. For example,

```
   (+ (if v e_1 e_2) e_big)

=  (let ((ctxt
           (λ (ret-val) (+ ret-val e_big))))
     (if v (ctxt e_1) (ctxt e_2)))
```

Full L4: begin

Transform **`begin`** expressions into **`let`** expressions using the rule

```
    (begin e1 e2)
 =  (let ((x e1)) e2)
```

which holds when **`x`** is not free in **`e2`**.

Full L4: multi-arg apps

Extend the `context` variant representing application expressions to accommodate multiple arguments by recording

- the sub-expressions that have already been normalized, and
- the sub-expressions remaining.

When none remain, rebuild the application and call `fill` as in the single argument case.

Full L4: primitive operators

`biop`, `pred`, and array/tuple expressions are like applications, if you pretend that the operator is a variable.

Full L4: variable freshness

The rules for lifting `let`s and eliminating `begin`s assume that the bound variable does not appear free in certain expressions.

Guarantee this constraint by giving every bound variable a fresh name in a pre-normalization pass.

Cleaning Up: copied code

In several `fill` cases, we repeated this pattern

> "If `d` is a value, do something; if it isn't, let it into a `let` and do the same thing to the `let`-bound variable."

Repetition is bad practice.

Cleaning Up: copied code

We should abstract over this pattern.

```
; maybe-let: L3-d (val → L3-e) → L3-e
(define (maybe-let d f)
  (if (val? d)
      (f d)
      (let ([x (fresh-var)])
        `(let ([,x ,d])
           ,(f x)))))
```

Cleaning Up: copied code

```
(define (fill d k)
  ...
  [fun-ctxt
   (a k)
   (maybe-let d
     (λ (v)
       (find a (arg-ctxt v k))))]
  ...)
```

Cleaning Up: copied code

```
(define (fill d k)
  ...
  [arg-ctxt
   (f k)
   (maybe-let d
     (λ (v)
       (fill `(,f ,v) k)))]
  ...)
```

Cleaning Up: copied code

```
(define (fill d k)
  (type-case context k
    ...
    [if-ctxt
     (t e k)
     (maybe-let d
       (λ (v)
         `(if ,v
              ,(find t k)
              ,(find e k))))]
    ...))
```

Cleaning Up: avoiding dispatch

Some calls to **`find`** add a layer to the context; calls to **`fill`** eventually remove the layer and switch on it.

Observe that for each call to **`find`**, we know which **`fill`** case eventually fires.

Cleaning Up: avoiding dispatch

```
(define (find e k)
  (match e
    ...
    [`(,f ,a)
     (find f (fun-ctxt a k))]
    ...))

(define (fill d k)
  ...
  [fun-ctxt
   (a k)
   (maybe-let d
     (λ (v)
       (find a (arg-ctxt v k))))]
  ...)
```

Cleaning Up: avoiding dispatch

```
(define (find e k)
  (match e
    ...
    [`(let ([,x ,r]) ,b)
     (find r (let-ctxt x b k))]
    ...))

(define (fill d k)
  (type-case context k
    ...
    [let-ctxt
     (x b k)
     `(let ([,x ,d])
        ,(find b k))]
    ...))
```

Cleaning Up: avoiding dispatch

```
(define (find e k)
  (match e
    ...
    [`(if ,c ,t ,e)
     (find c (if-ctxt t e k))]
    ...))

(define (fill d k)
  (type-case context k
    ...
    [if-ctxt
     (t e k)
     (maybe-let d
       (λ (v)
         `(if ,v
              ,(find t k)
              ,(find e k))))]
    ...))
```

Cleaning Up: avoiding dispatch

```
(define (fill d k)
  ...
  [fun-ctxt
   (a k)
   (maybe-let d
     (λ (v)
       (find a (arg-ctxt v k))))]
  ...
  [arg-ctxt
   (f k)
   (maybe-let d
     (λ (v)
       (fill `(,f ,v) k)))]
  ...)
```

Cleaning Up: avoiding dispatch

```
(define (norm e)
  (find e (no-ctxt)))

(define (fill d k)
  (type-case context k
    ...
    [no-ctxt () d]
    ...))
```

Cleaning Up: avoiding dispatch

Knowing which case eventually fires lets us eliminate the switch by replacing the `context` structures with functions that do whatever `fill` would do in the corresponding case.

The course website shows how to perform this refactoring as a sequence of small transformations.