Part 1

Defining a Language's Evaluation

`lambda.rkt`:

- With **`#lang plait`** ⇒ eager Curly
- With **`#lang plait #:lazy`** ⇒ lazy Curly

`more-lazy.rkt`:

- With **`#lang plait`** ⇒ lazy Curly
- With **`#lang plait #:lazy`** ⇒ lazy Curly

Let's make eager evaluation order explicit

Evaluation and “To do” Lists

```
(interp (plusE (numE 1) (numE 2)) mt-env)

⇒ (num+ (interp (numE 1) mt-env)
        (interp (numE 2) mt-env))

⇒ (interp (numE 1) mt-env)
```

To do:

```
(num+ ●
        (interp (numE 2) mt-env))
```

```
⇒ (numV 1)
```

To do:

```
(num+ ●
        (interp (numE 2) mt-env))
```

```
⇒ (interp (numE 2) mt-env)
```

To do:

```
(num+ (numV 1)
        ●)
```

```
⇒ (numV 2)
```

To do:

```
(num+ (numV 1)
        ●)
```

```
⇒ (num+ (numV 1) (numV 2))
```

Continuations

A “to do” list is a ***continuation***

To do:

```
(num+ ●
      (interp (numE 2) mt-env))
```

Continuations

A “to do” list is a ***continuation***

To do:

```
(+ 3
   (* ●
      (f (rest ls))))
```

A ***stack*** is one way to implement continuations

To do:

```
(* ● (f (rest ls)))
(+ 3 ●)
```

The terms ***stack*** and ***continuation*** are sometimes used interchangably

Part 2

Representing Continuations

To do:

```
{+ 3 ●}
```

```
(define-type Cont
  ....)
```

Representing Continuations

To do:

```
{+ 3 ●}
```

```
(define-type Cont
  (doPlusK [v : Value])
  ....)
```

```
  (doPlusK (numV 3))
```

Representing Continuations

To do:

```
{+ ● {f 0}}
```

```
(define-type Cont
  (doPlusK [v : Value])
  ....)
```

Representing Continuations

To do:
`{+ ● {f 0}}`

```
(define-type Cont
  (plusSecondK [r : Exp]
               [e : Env])
  (doPlusK [v : Value])
  ....)
```

```
(plusSecondK (appE (idE 'f) (numE 0))
             mt-env)
```

Representing Continuations

To do:

```
{+ ● {f 0}}
{+ 3 ●}
```

```
(define-type Cont
  (plusSecondK [r : Exp]
               [e : Env])
  (doPlusK [v : Value])
  ....)
```

Representing Continuations

To do:

```
{+ ● {f 0}}
{+ 3 ●}
```

```
(define-type Cont
  (plusSecondK [r : Exp]
               [e : Env]
               [k : Cont])
  (doPlusK [v : Value]
           [k : Cont])
  ....)
```

Representing Continuations

To do:

```
{+ ● {f 0}}
{+ 3 ●}
```

```
(define-type Cont
  (doneK)
  (plusSecondK [r : Exp]
               [e : Env]
               [k : Cont])
  (doPlusK [v : Value]
           [k : Cont])
  ....)
```

```
(plusSecondK (appE (idE 'f) (numE 0))
             mt-env
             (doPlusK (numV 3)
                      (doneK)))
```

Part 3

interp with Continuations

```
(define interp : (Exp Env -> Value)
  (lambda (a env)
    (type-case Exp a
      ...
      [(plusE l r) (num+ (interp l env)
                         (interp r env))]
      ...)))
```

interp with Continuations

```
(define interp : (Exp Env -> Value)
  (lambda (a env)
    (type-case Exp a
      ...
      [(plusE l r)  (interp l env)
                    (num+ ●
                          (interp r env))]
      ...)))
```

interp with Continuations

```
(define interp : (Exp Env -> Value)
  (lambda (a env)
    (type-case Exp a
      ...
      [(plusE l r) (interp l env)
                   (num+ ●
                         (interp r env))]
      ...)))
```

To do:

`{+ ● <Exp>}`

interp with Continuations

```
(define interp : (Exp Env -> Value)
  (lambda (a env)
    (type-case Exp a
      ...
      [(plusE l r) (interp l env)
                   (num+ ●
                         (interp r env))]
      ...)))
```

To do:

```
(num+ ●
      (interp r env))
```

interp with Continuations

```
(define interp : (Exp Env -> Value)
  (lambda (a env)
    (type-case Exp a
      ...
      [(plusE l r) (interp l env)
                   (plusSecondK r env)]
      ...)))
```

To do:

```
(num+ ●
      (interp r env))
```

interp with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(plusE l r)  (interp l env)
                    (plusSecondK r env k)]
      ...)))
```

To do:

```
(num+ ●
      (interp r env))
....
```

interp with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(plusE l r) (interp l env)
                   (plusSecondK r env k)]
      ...)))
```

interp with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(plusE l r) (interp l env
                           (plusSecondK r env k))]
      ...)))
```

interp with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(numE n) (numV n)]
     ...)))
```

interp with Continuations

```
(define interp : (Exp Env Cont -> Value)
 (lambda (a env k)
  (type-case Exp a
   ...
   [(numE n) (continue k (numV n))]
  ...)))
```

interp with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(numE n) (continue k (numV n))]
     ...)))

(define continue : (Cont Value -> Value)
  (lambda (k v)
    (type-case Cont k
      ...

      ...)))
```

interp with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(numE n) (continue k (numV n))]
     ...)))

(define continue : (Cont Value -> Value)
  (lambda (k v)
    (type-case Cont k
      ...
      [(doneK) v]
      ...)))
```

interp with Continuations

```
(define interp : (Exp Env Cont -> Value)
 (lambda (a env k)
  (type-case Exp a
   ...
   [(numE n) (continue k (numV n))]
  ...)))

(define continue : (Cont Value -> Value)
 (lambda (k v)
  (type-case Cont k
   ...
   [(plusSecondK r env next-k)
    (interp r env
            ...)]
   ...)))
```

interp with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(numE n) (continue k (numV n))]
     ...)))

(define continue : (Cont Value -> Value)
  (lambda (k v)
    (type-case Cont k
      ...
      [(plusSecondK r env next-k)
       (interp r env
               ...)]
      ...)))
```

To do:

```
(num+ ●
      (interp r env))
....
```

interp with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(numE n) (continue k (numV n))]
     ...)))

(define continue : (Cont Value -> Value)
  (lambda (k v)
    (type-case Cont k
      ...
      [(plusSecondK r env next-k)
       (interp r env
               ...)]
      ...)))
```

To do:

```
(num+ v
      ●)
....
```

`interp` with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(numE n) (continue k (numV n))]
     ...)))

(define continue : (Cont Value -> Value)
  (lambda (k v)
    (type-case Cont k
      ...
      [(plusSecondK r env next-k)
       (interp r env
               (doPlusK v next-k))]
      ...)))
```

To do:

```
(num+ v
        ●)
....
```

interp with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(numE n) (continue k (numV n))]
     ...)))

(define continue : (Cont Value -> Value)
  (lambda (k v)
    (type-case Cont k
      ...
      [(doPlusK v-l next-k)
       (continue next-k (num+ v-l v))]
      ...)))
```

To do:

```
(num+ v-l
        ●)
....
```

Part 4

interp with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(numE n) (continue k (numV n))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(doneK) v]
    ...))
```

```
(interp (numE 5) mt-env (doneK))

⇒ (continue (doneK) (numV 5))

⇒ (numV 5)
```

interp with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(plusE l r) (interp l env (plusSecondK r env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(plusSecondK r env next-k)
     (interp r env (doPlusK v next-k))]
    [(doPlusK v-l next-k)
     (continue next-k (num+ v-l v))]
    ...))
```

```
(interp (plusE (numE 5) (numE 2)) mt-env (doneK))

⇒ (interp (numE 5)
          (plusSecondK (numE 2) mt-env (doneK)))

⇒ (continue (plusSecondK (numE 2) mt-env (doneK))
            (numV 5))
```

190–193

interp with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(plusE l r) (interp l env (plusSecondK r env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(plusSecondK r env next-k)
     (interp r env (doPlusK v next-k))]
    [(doPlusK v-l next-k)
     (continue next-k (num+ v-l v))]
    ...))
```

```
⇒ (continue (plusSecondK (numE 2) mt-env (doneK))
            (numV 5))

⇒ (interp (numE 2) mt-env
          (doPlusK (numV 5) (doneK)))

⇒ (continue (doPlusK (numV 5) (doneK))
            (numV 2))
```

interp with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(plusE l r) (interp l env (plusSecondK r env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(plusSecondK r env next-k)
     (interp r env (doPlusK v next-k))]
    [(doPlusK v-l next-k)
     (continue next-k (num+ v-l v))]
    ...))
```

```
⇒ (continue (doPlusK (numV 5) (doneK))
            (numV 2))

⇒ (continue (doneK)
            (numV 7))
```

Part 5

interp with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(lamE n body)
     (continue k (closV n body env))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...

    ...))
```

interp with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(appE fun arg) (interp fun env (appArgK arg env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(appArgK a env next-k)
     (interp a env (doAppK v next-k))]
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV n body c-env)
        (interp body (extend-env
                      (bind n v)
                      c-env)
                next-k)]
       [else (error ...)])]
    ...))
```

interp with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(appE fun arg) (interp fun env (appArgK arg env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(appArgK a env next-k)
     (interp a env (doAppK v next-k))]
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV n body c-env)
        (interp body (extend-env
                      (bind n v)
                      c-env)
                next-k)]
       [else (error ...)])]
    ...))
```

interp with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(appE fun arg) (interp fun env (appArgK arg env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(appArgK a env next-k)
     (interp a env (doAppK v next-k))]
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV n body c-env)
        (interp body (extend-env
                      (bind n v)
                      c-env)
                next-k)]
       [else (error ...)])]
    ...))
```

```
E1 = (extend-env
      (bind 'f (closV 'x
                      (idE 'x)
                      mt-env))
      mt-env)
```

```
(interp (appE (idE 'f) (numE 1))
        E1
        (doneK))

⇒ (interp (idE 'f)
          E1
          (appArgK (numE 1) E1 (doneK)))
```

interp with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(appE fun arg) (interp fun env (appArgK arg env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(appArgK a env next-k)
     (interp a env (doAppK v next-k))]
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV n body c-env)
        (interp body (extend-env
                      (bind n v)
                      c-env)
                next-k)]
       [else (error ...)])]
    ...))
```

```
$E_1$ = (extend-env
          (bind 'f (closV 'x
                          (idE 'x)
                          mt-env))
          mt-env)
```

```
⇒ (interp (idE 'f)
          $E_1$
          (appArgK (numE 1) $E_1$ (doneK)))

⇒ (continue (appArgK (numE 1) $E_1$ (doneK))
            (closV 'x (idE 'x) mt-env))
```

`interp` with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(appE fun arg) (interp fun env (appArgK arg env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(appArgK a env next-k)
     (interp a env (doAppK v next-k))]
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV n body c-env)
        (interp body (extend-env
                      (bind n v)
                      c-env)
                next-k)]
       [else (error ...)])]
    ...))
```

```
E_1 = (extend-env
      (bind 'f (closV 'x
                      (idE 'x)
                      mt-env))
      mt-env)
```

```
⇒ (continue (appArgK (numE 1) E_1 (doneK))
            (closV 'x (idE 'x) mt-env))

⇒ (interp (numE 1)
          E_1
          (doAppK (closV 'x (idE 'x) mt-env) (doneK)))
```

interp with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(appE fun arg) (interp fun env (appArgK arg env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(appArgK a env next-k)
     (interp a env (doAppK v next-k))]
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV n body c-env)
        (interp body (extend-env
                      (bind n v)
                      c-env)
                next-k)]
       [else (error ...)])]
    ...))
```

```
E_1 = (extend-env
        (bind 'f (closV 'x
                        (idE 'x)
                        mt-env))
        mt-env)
```

```
⇒ (interp (numE 1)
          E_1
          (doAppK (closV 'x (idE 'x) mt-env) (doneK)))

⇒ (continue (doAppK (closV 'x (idE 'x) mt-env) (doneK))
            (numV 1))
```

interp with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(appE fun arg) (interp fun env (appArgK arg env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(appArgK a env next-k)
     (interp a env (doAppK v next-k))]
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV n body c-env)
        (interp body (extend-env
                      (bind n v)
                      c-env)
                next-k)]
       [else (error ...)])]
    ...))
```

```
E1 = (extend-env
       (bind 'f (closV 'x
                       (idE 'x)
                       mt-env))
       mt-env)
```

```
⇒ (continue (doAppK (closV 'x (idE 'x) mt-env) (doneK))
            (numV 1))

⇒ (interp (idE 'x)
          (extend-env (bind 'x (numV 1)) mt-env)
          (doneK))
```

interp with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(appE fun arg) (interp fun env (appArgK arg env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(appArgK a env next-k)
     (interp a env (doAppK v next-k))]
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV n body c-env)
        (interp body (extend-env
                      (bind n v)
                      c-env)
                next-k)]
       [else (error ...)])]
    ...))
```

```
E1 = (extend-env
      (bind 'f (closV 'x
                      (idE 'x)
                      mt-env))
      mt-env)
```

```
⇒ (interp (idE 'x)
          (extend-env (bind 'x (numV 1)) mt-env)
          (doneK))

⇒ (continue (doneK)
            (numV 1))
```

Part 6

Infinite Loop

```
while (1) { }
```

Space-Bounded Loop

```
int f() { return f(); }
```

Curly Loop

```
{let {[f {lambda {f}
              {f f}}]}
  {f f}}
```

infinite or space-bounded?

Curly Loop

```
{let {[f {lambda {f} {f f}}]}
  {f f}}
```

Curly Loop

```
{let {[f {lambda {f} {f f}}]}
  {f f}}
⇒
{{lambda {f} {f f}}
 {lambda {f} {f f}}}
```

Curly Loop

```
{{lambda {f} {f f}}
 {lambda {f} {f f}}}
```

Curly Loop

```
{{lambda {f} {f f}}
 {lambda {f} {f f}}}
⇒
{{lambda {f} {f f}}
 {lambda {f} {f f}}}
```

Curly Loop

```
(interp {{lambda {f} {f f}}
          {lambda {f} {f f}}}
        mt-env
        (doneK))
```

Curly Loop

```
(interp {{lambda {f} {f f}}
         {lambda {f} {f f}}}
        mt-env
        (doneK))
⇒
(interp {lambda {f} {f f}}
        mt-env
        (appArgK {lambda {f} {f f}}
                 mt-env
                 (doneK)))
```

Curly Loop

```
(interp {lambda {f} {f f}}
        mt-env
        (appArgK {lambda {f} {f f}}
                 mt-env
                 (doneK)))
```

Curly Loop

```
(interp {lambda {f} {f f}}
        mt-env
        (appArgK {lambda {f} {f f}}
                 mt-env
                 (doneK)))
⇒
(continue (appArgK {lambda {f} {f f}}
                   mt-env
                   (doneK))
          (closV 'f {f f} mt-env))
```

Curly Loop

```
(continue (appArgK {lambda {f} {f f}}
                   mt-env
                   (doneK))
          (closV 'f {f f} mt-env))
```

Curly Loop

```
(continue (appArgK {lambda {f} {f f}}
                   mt-env
                   (doneK))
          (closV 'f {f f} mt-env))
⇒
(interp {lambda {f} {f f}}
        mt-env
        (doAppK (closV 'f {f f} mt-env)
                (doneK)))
```

Curly Loop

```
(interp {lambda {f} {f f}}
        mt-env
        (doAppK (closV 'f {f f} mt-env)
                (doneK)))
```

Curly Loop

```
(interp {lambda {f} {f f}}
        mt-env
        (doAppK (closV 'f {f f} mt-env)
                (doneK)))
⇒
(continue (doAppK (closV 'f {f f} mt-env)
                  (doneK))
          (closV 'f {f f} mt-env))
```

Curly Loop

```
(continue (doAppK (closV 'f {f f} mt-env)
                  (doneK))
          (closV 'f {f f} mt-env))
```

Curly Loop

```
(continue (doAppK (closV 'f {f f} mt-env)
                  (doneK))
          (closV 'f {f f} mt-env))
⇒
(interp {f f}
        (extend-env
         (bind 'f (closV 'f {f f} mt-env))
        mt-env)
        (doneK))
```

Curly Loop

```
(interp {f f}
        (extend-env
         (bind 'f (closV 'f {f f} mt-env))
         mt-env)
        (doneK))
```

Curly Loop

```
(interp {f f}
        (extend-env
         (bind 'f (closV 'f {f f} mt-env))
         mt-env)
        (doneK))
```

E_1

Curly Loop

```
(interp {f f}
        (extend-env
  E1     (bind 'f (closV 'f {f f} mt-env))
         mt-env)
        (doneK))
=
(interp {f f}
        E1
        (doneK))
```

Curly Loop

```
(interp {f f}
        E1
        (doneK))
```

Curly Loop

```
(interp {f f}
        E_1
         (doneK))
⇒
(interp f
        E_1
         (appArgK f  E_1
                   (doneK)))
```

Curly Loop

```
(interp f
        E1
        (appArgK f E1
                 (doneK)))
```

Curly Loop

```
(interp f
        E1
        (appArgK f E1
                 (doneK)))
⇒
(continue (appArgK f E1
                   (doneK))
          (closV 'f {f f} mt-env))
```

Curly Loop

```
(continue (appArgK f E1
                   (doneK))
          (closV 'f {f f} mt-env))
```

Curly Loop

```
(continue (appArgK f E1
                   (doneK))
          (closV 'f {f f} mt-env))
⇒
(interp f
        E1
        (doAppK (closV 'f {f f} mt-env)
                (doneK)))
```

Curly Loop

```
(interp f
        E1
        (doAppK (closV 'f {f f} mt-env)
                (doneK)))
```

Curly Loop

```
(interp f
        E1
        (doAppK (closV 'f {f f} mt-env)
                (doneK)))
⇒
(continue (doAppK (closV 'f {f f} mt-env)
                  (doneK))
          (closV 'f {f f} mt-env))
```

Curly Loop

```
(continue (doAppK (closV 'f {f f} mt-env)
                  (doneK))
          (closV 'f {f f} mt-env))
```

Curly Loop

```
(continue (doAppK (closV 'f {f f} mt-env)
                  (doneK))
          (closV 'f {f f} mt-env))
⇒
(interp {f f}
        E1
         (doneK))
```

Part 7

Curly Loop?

```
{let {[f {lambda {f} {+ 1 {f f}}}]}
  {f f}}
```

Curly Loop?

```
{let {[f {lambda {f} {+ 1 {f f}}}]}
  {f f}}
⇒
{{lambda {f} {+ 1 {f f}}}
 {lambda {f} {+ 1 {f f}}}}
```

Curly Loop?

```
{{lambda {f} {+ 1 {f f}}}
 {lambda {f} {+ 1 {f f}}}}
```

Curly Loop?

```
{{lambda {f} {+ 1 {f f}}}
 {lambda {f} {+ 1 {f f}}}}
⇒
{+ 1 {{lambda {f} {+ 1 {f f}}}
      {lambda {f} {+ 1 {f f}}}}}
```

Curly Loop?

```
{+ 1 {{lambda {f} {+ 1 {f f}}}
      {lambda {f} {+ 1 {f f}}}}}
```

Curly Loop?

```
{+ 1 {{lambda {f} {+ 1 {f f}}}
      {lambda {f} {+ 1 {f f}}}}}
⇒
{+ 1 {+ 1 {{lambda {f} {+ 1 {f f}}}
           {lambda {f} {+ 1 {f f}}}}}}
```

Tail Calls

```
(define (forever x)
  (forever (not x)))
```

Call to `forever` is a ***tail call***, because there’s no work to do after `forever` returns

Non-Tail Calls

```
(define (run-out-of-memory x)
  (not (run-out-of-memory x)))
```

The call to `run-out-of-memory` is *not* a tail call, because there's work to do after it returns

Tail Calls

```
(define (forever x)
  (if x
      (forever #t)
      (forever #f)))
```

Even though the call to `forever` is wrapped in `if`, there's no work to do after `forever` returns

The branches of `if` are in ***tail position*** with respect to the `if`

Non-Tail Calls

```
(define (run-out-of-memory x)
  (if (run-out-of-memory x)
      #t
      #f))
```

The call to `run-out-of-memory` is *not* a tail call, because there's work to do after it returns

The test position `if` is *not* in tail position with respect to the `if`

`interp` and `continue`

In **`lambda-k.rkt`**:

- `interp` calls `continue` only as a tail call
- `continue` calls `interp` only as a tail call
- `lookup` calls `lookup` only as a tail call
- nothing else is recursive

∴ the Plait continuation is always small