Part I

Defining a Language's Evaluation

`lambda.rhm`:

- **`#lang shplait`** ⇒ eager Moe
- **`#lang shplait`** with **`~lazy`** ⇒ lazy Moe

`more_lazy.rhm`:

- With **`#lang shplait`** ⇒ lazy Moe
- With **`#lang shplait`** with **`~lazy`** ⇒ lazy Moe

Let's make eager evaluation order explicit

Evaluation and "To do" Lists

```
interp(plusE(intE(1), intE(2)), mt_env)

⇒ num_plus(interp(intE(1), mt_env),
           interp(intE(2), mt_env))

⇒ interp(intE(1), mt_env)
```

To do:
`num_plus(●, interp(intE(2), mt_env))`

```
⇒ intV(1)
```

To do:
`num_plus(●, interp(intE(2), mt_env))`

```
⇒ interp(intE(2), mt_env)
```

To do:
`num_plus(intV(1), ●)`

```
⇒ intV(2)
```

To do:
`num_plus(intV(1), ●)`

```
⇒ num_plus(intV(1), intV(2))
```

Continuations

A "to do" list is a ***continuation***

To do:

```
num_plus(●,
         interp(intE(2), mt_env))
```

Continuations

A "to do" list is a ***continuation***

To do:
```
3 + ● * f(rest(ls))
```

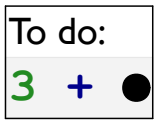
A ***stack*** is one way to implement continuations

To do:
```
● * f(rest(ls))
3 + ●
```

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

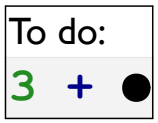
Part 2

Representing Continuations




```
type Cont
....
```

Representing Continuations




```
type Cont
| doPlusK(v :: Value)
....

   doPlusK(intV(3))
```

Representing Continuations

To do:

- `● + f(0)`

```
type Cont
| doPlusK(v :: Value)
....
```

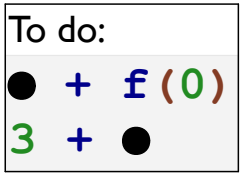
Representing Continuations

To do:

- `+ f(0)`

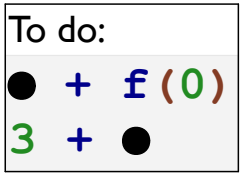
```
type Cont
| plusSecondK(r :: Exp,
               e :: Env)
| doPlusK(v :: Value)
....

plusSecondK(appE(idE(#'f), intE(0)),
            mt_env)
```

Representing Continuations




```
type Cont
| plusSecondK(r :: Exp,
               e :: Env)
| doPlusK(v :: Value)
....
```

Representing Continuations




```
type Cont
| plusSecondK(r :: Exp,
              e :: Env,
              k :: Cont)
| doPlusK(v :: Value,
          k :: Cont)
....
```

Representing Continuations

To do:

● + f(0)

3 + ●

```
type Cont
| doneK()
| plusSecondK(r :: Exp,
              e :: Env,
              k :: Cont)
| doPlusK(v :: Value,
          k :: Cont)
....
```

```
plusSecondK(appE(idE(#'f), intE(0)),
            mt_env,
            doPlusK(intV(3),
                    doneK()))
```

Part 3

`interp` with Continuations

```
def interp :: (Exp, Env) -> Value:
  fun (a, env):
    match a
    | ....
    | plusE(l, r): num_plus(interp(l, env),
                            interp(r, env))
    | ....
```

`interp` with Continuations

```
def interp :: (Exp, Env) -> Value:
  fun (a, env):
    match a
    | ....
    | plusE(l, r): interp(l, env)
                   num_plus(●                 ,
                            interp(r, env))
    | ....
```

`interp` with Continuations

```
def interp :: (Exp, Env) -> Value:
  fun (a, env):
    match a
    | ....
    | plusE(l, r): interp(l, env)
                   num_plus(●                 ,
                            interp(r, env))
    | ....
```

To do:

● + <Exp>

`interp` with Continuations

```
def interp :: (Exp, Env) -> Value:
  fun (a, env):
    match a
    | ....
    | plusE(l, r): interp(l, env)
                   num_plus(●                 ,
                            interp(r, env))
    | ....
```

To do:

```
num_plus(●,
         interp(r, env))
```

`interp` with Continuations

```
def interp :: (Exp, Env) -> Value:
  fun (a, env):
    match a
    | ....
    | plusE(l, r): interp(l, env)
                   plusSecondK(r, env)
    | ....
```

To do:

```
num_plus(●,
         interp(r, env))
```

`interp` with Continuations

```
def interp :: (Exp, Env, Cont) -> Value:
  fun (a, env, k):
    match a
    | ....
    | plusE(l, r): interp(l, env)
                   plusSecondK(r, env, k)
    | ....
```

To do:

```
num_plus(●,
         interp(r, env))
....
```

`interp` with Continuations

```
def interp :: (Exp, Env, Cont) -> Value:
  fun (a, env, k):
    match a
    | ....
    | plusE(l, r): interp(l, env)
                   plusSecondK(r, env, k)
    | ....
```

`interp` with Continuations

```
def interp :: (Exp, Env, Cont) -> Value:
  fun (a, env, k):
    match a
    | ....
    | plusE(l, r): interp(l, env, plusSecondK(r, env, k))
    | ....
```

`interp` with Continuations

```
def interp :: (Exp, Env, Cont) -> Value:
  fun (a, env, k):
    match a
    | ....
    | intE(n): intV(n)
    | ....
```

`interp` with Continuations

```
def interp :: (Exp, Env, Cont) -> Value:
  fun (a, env, k):
    match a
    | ....
    | intE(n): continue(k, intV(n))
    | ....
```

`interp` with Continuations

```
def interp :: (Exp, Env, Cont) -> Value:
  fun (a, env, k):
    match a
    | ....
    | intE(n): continue(k, intV(n))
    | ....

def continue :: (Cont, Value) -> Value:
  fun (k, v):
    match k
    | ....

    | ....
```

`interp` with Continuations

```
def interp :: (Exp, Env, Cont) -> Value:
  fun (a, env, k):
    match a
    | ....
    | intE(n): continue(k, intV(n))
    | ....

def continue :: (Cont, Value) -> Value:
  fun (k, v):
    match k
    | ....
    | doneK(): v
    | ....
```

`interp` with Continuations

```
def interp :: (Exp, Env, Cont) -> Value:
  fun (a, env, k):
    match a
    | ....
    | intE(n): continue(k, intV(n))
    | ....

def continue :: (Cont, Value) -> Value:
  fun (k, v):
    match k
    | ....
    | plusSecondK(r, env, next_k):
        interp(r, env,
               ....)
    | ....
```

`interp` with Continuations

```
def interp :: (Exp, Env, Cont) -> Value:
  fun (a, env, k):
    match a
    | ....
    | intE(n): continue(k, intV(n))
    | ....

def continue :: (Cont, Value) -> Value:
  fun (k, v):
    match k
    | ....
    | plusSecondK(r, env, next_k):
        interp(r, env,
               ....)
    | ....
```

To do:

```
num_plus(●,
         interp(r, env))
....
```

`interp` with Continuations

```
def interp :: (Exp, Env, Cont) -> Value:
  fun (a, env, k):
    match a
    | ....
    | intE(n): continue(k, intV(n))
    | ....

def continue :: (Cont, Value) -> Value:
  fun (k, v):
    match k
    | ....
    | plusSecondK(r, env, next_k):
        interp(r, env,
               ....)
    | ....
```

To do:

```
num_plus(v, ●)
....
```

`interp` with Continuations

```
def interp :: (Exp, Env, Cont) -> Value:
  fun (a, env, k):
    match a
    | ....
    | intE(n): continue(k, intV(n))
    | ....

def continue :: (Cont, Value) -> Value:
  fun (k, v):
    match k
    | ....
    | plusSecondK(r, env, next_k):
        interp(r, env,
               doPlusK(v, next_k))
    | ....
```

To do:

```
num_plus(v, ●)
....
```

`interp` with Continuations

```
def interp :: (Exp, Env, Cont) -> Value:
  fun (a, env, k):
    match a
    | ....
    | intE(n): continue(k, intV(n))
    | ....

def continue :: (Cont, Value) -> Value:
  fun (k, v):
    match k
    | ....
    | doPlusK(v_l, next_k):
        continue(next_k,
                 num_plus(v_l, v))
    | ....
```

To do:

```
num_plus(v_l, ●)
....
```

Part 4

`interp` with Continuations

```
fun interp(a :: Exp, env :: Env, k :: Cont) :: Value:
  match a ....
  | intE(n): continue(k, intV(n))
  | ....

fun continue(k :: Cont, v :: Value) :: Value:
  match k ....
  | doneK(): v
  | ....
```

```
interp(intE(5), mt_env, doneK())

⇒ continue(doneK(), intV(5))

⇒ intV(5)
```

`interp` with Continuations

```
fun interp(a :: Exp, env :: Env, k :: Cont) :: Value:
  match a ....
  | plusE(l, r): interp(l, env, plusSecondK(r, env, k))
  | ....

fun continue(k :: Cont, v :: Value) :: Value:
  match k ....
  | plusSecondK(r, env, next_k):
      interp(r, env, doPlusK(v, next_k))
  | doPlusK(v_l, next_k):
      continue(next_k, num_plus(v_l, v))
  | ....
```

```
interp(plusE(intE(5), intE(2)), mt_env, doneK())

⇒ interp(intE(5), mt_env,
         plusSecondK(intE(2), mt_env, doneK()))

⇒ continue(plusSecondK(intE(2), mt_env, doneK()),
           intV(5))
```

`interp` with Continuations

```
fun interp(a :: Exp, env :: Env, k :: Cont) :: Value:
  match a ....
  | plusE(l, r): interp(l, env, plusSecondK(r, env, k))
  | ....

fun continue(k :: Cont, v :: Value) :: Value:
  match k ....
  | plusSecondK(r, env, next_k):
      interp(r, env, doPlusK(v, next_k))
  | doPlusK(v_l, next_k):
      continue(next_k, num_plus(v_l, v))
  | ....
```

```
⇒ continue(plusSecondK(intE(2), mt_env, doneK()),
           intV(5))

⇒ interp(intE(2), mt_env,
         doPlusK(intV(5), doneK()))

⇒ continue(doPlusK(intV(5), doneK()),
           intV(2))
```

`interp` with Continuations

```
fun interp(a :: Exp, env :: Env, k :: Cont) :: Value:
  match a ....
  | plusE(l, r): interp(l, env, plusSecondK(r, env, k))
  | ....

fun continue(k :: Cont, v :: Value) :: Value:
  match k ....
  | plusSecondK(r, env, next_k):
      interp(r, env, doPlusK(v, next_k))
  | doPlusK(v_l, next_k):
      continue(next_k, num_plus(v_l, v))
  | ....
```

```
⇒ continue(doPlusK(intV(5), doneK()),
           intV(2))

⇒ continue(doneK(),
           intV(7))
```

Part 5

`interp` with Continuations

```
fun interp(a :: Exp, env :: Env, k :: Cont) :: Value:
  match a ....
  | funE(n, body):
      continue(k, closV(n, body, env))
  | ....

fun continue(k :: Cont, v :: Value) :: Value:
  match k ....

  | ....
```

`interp` with Continuations

```
fun interp(a :: Exp, env :: Env, k :: Cont) :: Value:
  match a ....
  | appE(fn, arg): interp(fn, env, appArgK(arg, env, k))
  | ....

fun continue(k :: Cont, v :: Value) :: Value:
  match k ....
  | appArgK(a, env, next_k):
      interp(a, env, doAppK(v, next_k))
  | doAppK(v_f, next_k):
      match v_f
      | closV(n, body, c_env):
          interp(body, extend_env(bind(n, v),
                                  c_env),
                 next_k)
      | ~else: error(....)
  | ....
```

`interp` with Continuations

```
fun interp(a :: Exp, env :: Env, k :: Cont) :: Value:
  match a ....
  | appE(fn, arg): interp(fn, env, appArgK(arg, env, k))
  | ....

fun continue(k :: Cont, v :: Value) :: Value:
  match k ....
  | appArgK(a, env, next_k):
      interp(a, env, doAppK(v, next_k))
  | doAppK(v_f, next_k):
      match v_f
      | closV(n, body, c_env):
          interp(body, extend_env(bind(n, v),
                                  c_env),
                 next_k)
      | ~else: error(....)
  | ....
```

`interp` with Continuations

```
fun interp(a :: Exp, env :: Env, k :: Cont) :: Value:
  match a ....
  | appE(fn, arg): interp(fn, env, appArgK(arg, env, k))
  | ....

fun continue(k :: Cont, v :: Value) :: Value:
  match k ....
  | appArgK(a, env, next_k):
      interp(a, env, doAppK(v, next_k))
  | doAppK(v_f, next_k):
      match 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(appE(idE(#'f), intE(1)),
       E1,
       doneK())

⇒ interp(idE(#'f),
         E1,
         appArgK(intE(1), E1, doneK()))
```

`interp` with Continuations

```
fun interp(a :: Exp, env :: Env, k :: Cont) :: Value:
  match a ....
  | appE(fn, arg): interp(fn, env, appArgK(arg, env, k))
  | ....

fun continue(k :: Cont, v :: Value) :: Value:
  match k ....
  | appArgK(a, env, next_k):
      interp(a, env, doAppK(v, next_k))
  | doAppK(v_f, next_k):
      match 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), E1, appArgK(intE(1), E1, doneK()))`

⇒ `continue(appArgK(intE(1), E1, doneK()), closV(#'x, idE(#'x), mt_env))`

`interp` with Continuations

```
fun interp(a :: Exp, env :: Env, k :: Cont) :: Value:
  match a ....
  | appE(fn, arg): interp(fn, env, appArgK(arg, env, k))
  | ....

fun continue(k :: Cont, v :: Value) :: Value:
  match k ....
  | appArgK(a, env, next_k):
      interp(a, env, doAppK(v, next_k))
  | doAppK(v_f, next_k):
      match 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(intE(1), E1, doneK()), closV(#'x, idE(#'x), mt_env))`

⇒ `interp(intE(1), E1, doAppK(closV(#'x, idE(#'x), mt_env), doneK()))`

`interp` with Continuations

```
fun interp(a :: Exp, env :: Env, k :: Cont) :: Value:
  match a ....
  | appE(fn, arg): interp(fn, env, appArgK(arg, env, k))
  | ....

fun continue(k :: Cont, v :: Value) :: Value:
  match k ....
  | appArgK(a, env, next_k):
      interp(a, env, doAppK(v, next_k))
  | doAppK(v_f, next_k):
      match 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(intE(1), `E_1`, doAppK(closV(#'x, idE(#'x), mt_env), doneK()))`

⇒ `continue(doAppK(closV(#'x, idE(#'x), mt_env), doneK()), intV(1))`

`interp` with Continuations

```
fun interp(a :: Exp, env :: Env, k :: Cont) :: Value:
  match a ....
  | appE(fn, arg): interp(fn, env, appArgK(arg, env, k))
  | ....

fun continue(k :: Cont, v :: Value) :: Value:
  match k ....
  | appArgK(a, env, next_k):
      interp(a, env, doAppK(v, next_k))
  | doAppK(v_f, next_k):
      match 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(doAppK(closV(#'x, idE(#'x), mt_env), doneK()), intV(1))`

⇒ `interp(idE(#'x), extend_env(bind(#'x, intV(1)), mt_env), doneK())`

`interp` with Continuations

```
fun interp(a :: Exp, env :: Env, k :: Cont) :: Value:
  match a ....
  | appE(fn, arg): interp(fn, env, appArgK(arg, env, k))
  | ....

fun continue(k :: Cont, v :: Value) :: Value:
  match k ....
  | appArgK(a, env, next_k):
      interp(a, env, doAppK(v, next_k))
  | doAppK(v_f, next_k):
      match 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(#'x),
       extend_env(bind(#'x, intV(1)), mt_env),
       doneK())
```

⇒
```
continue(doneK(),
         intV(1))
```

Part 6

Loop Syntax versus Function Calls

The Java way:

```
int sum(List<Integer> lst) {
  int t = 0;
  for (Integer n : lst) {
    t = t + n;
  }
  return t;
}
```

The Shplait way:

```
fun sum(lst :: Listof(Int), t :: Int):
  match lst
  | []: t
  | cons(f, rst): sum(rst, f + t)
```

Infinite Loop

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

Space-Bounded Loop

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

Moe Loop

```
let f = (fun (f):
          f(f)):
  f(f)
```

infinite or space-bounded?

Moe Loop

```
let f = (fun (f): f(f)):
  f(f)
```

Moe Loop

```
let f = (fun (f): f(f)):
  f(f)
```

⇒

```
(fun (f): f(f))(fun (f): f(f))
```

Moe Loop

```
(fun (f): f(f))(fun (f): f(f))
```

Moe Loop

```
(fun (f): f(f))(fun (f): f(f))
```

⇒

```
(fun (f): f(f))(fun (f): f(f))
```

Moe Loop

```
interp((fun (f): f(f))(fun (f): f(f)),
       mt_env,
       doneK())
```

Moe Loop

```
interp((fun (f): f(f))(fun (f): f(f)),
       mt_env,
       doneK())
```

need several setup steps...

Moe Loop

```
interp((fun (f): f(f))(fun (f): f(f)),
       mt_env,
       doneK())
⇒
interp(fun (f): f(f),
       mt_env,
       appArgK(fun (f): f(f),
               mt_env,
               doneK()))
```

Moe Loop

```
interp(fun (f): f(f),
       mt_env,
       appArgK(fun (f): f(f),
               mt_env,
               doneK()))
```

Moe Loop

```
interp(fun (f): f(f),
       mt_env,
       appArgK(fun (f): f(f),
               mt_env,
               doneK()))
```

⇒

```
continue(appArgK(fun (f): f(f),
                 mt_env,
                 doneK()),
         closV(#'f, f(f), mt_env))
```

Moe Loop

```
continue(appArgK(fun (f): f(f),
                 mt_env,
                 doneK()),
         closV(#'f, f(f), mt_env))
```

Moe Loop

```
continue(appArgK(fun (f): f(f),
                 mt_env,
                 doneK()),
         closV(#'f, f(f), mt_env))
```

⇒

```
interp(fun (f): f(f),
       mt_env,
       doAppK(closV(#'f, f(f), mt_env),
              doneK()))
```

Moe Loop

```
interp(fun (f): f(f),
       mt_env,
       doAppK(closV(#'f, f(f), mt_env),
              doneK()))
```

Moe Loop

```
interp(fun (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))
```

Moe Loop

```
continue(doAppK(closV(#'f, f(f), mt_env),
                doneK()),
         closV(#'f, f(f), mt_env))
```

Moe 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())
```

Moe Loop

```
interp(f(f),
       extend_env(bind(#'f, closV(#'f,
                                  f(f),
                                  mt_env)),
                  mt_env),
       doneK())
```

Moe Loop

```
interp( f(f) ,
        extend_env(bind(#'f, closV(#'f,
                                   f(f) ,
                                   mt_env)),
                   mt_env),
        doneK())
```

E_1

Moe Loop

```
interp(f(f),
       extend_env(bind(#'f, closV(#'f,
                                  f(f),
                                  mt_env)),
                  mt_env),
       doneK())
=
interp(f(f),
       E1,
       doneK())
```

E_1

Moe Loop

```
interp(f(f),
       extend_env(bind(#'f, closV(#'f,
                                  f(f),
                                  mt_env)),
                  mt_env),
       doneK())
=
interp(f(f),
       E1,
       doneK())
```

E_1

this is the key state

Moe Loop

```
interp( f(f) ,
        E1,
        doneK() )
```

Moe Loop

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

Moe Loop

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

Moe Loop

```
interp(f,
       E1,
       appArgK(f, E1,
               doneK()))
⇒
continue(appArgK(f, E1,
                 doneK()),
         closV(#'f, f(f), mt_env))
```

Moe Loop

```
continue(appArgK(f, E1,
                 doneK()),
         closV(#'f, f(f), mt_env))
```

Moe Loop

```
continue(appArgK(f, E1,
                 doneK()),
         closV(#'f, f(f), mt_env))
⇒
interp(f,
       E1,
       doAppK(closV(#'f, f(f), mt_env),
              doneK()))
```

Moe Loop

```
interp(f,
       E1,
       doAppK(closV(#'f, f(f), mt_env),
              doneK()))
```

Moe 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))
```

Moe Loop

```
continue(doAppK(closV(#'f, f(f), mt_env),
                doneK()),
         closV(#'f, f(f), mt_env))
```

Moe Loop

```
continue(doAppK(closV(#'f, f(f), mt_env),
                doneK()),
         closV(#'f, f(f), mt_env))
⇒
interp(f(f),
       E1,
       doneK())
```

Moe Loop

```
continue(doAppK(closV(#'f, f(f), mt_env),
                doneK()),
         closV(#'f, f(f), mt_env))
```

⇒

```
interp(f(f),
       E1,
       doneK())
```

back to key state

⇒ loops forever

Part 7

Moe Loop?

```
let f = (fun (f): 1 + f(f)):
  f(f)
```

Moe Loop?

```
let f = (fun (f): 1 + f(f)):
  f(f)
```

⇒

```
(fun (f): 1 + f(f))(fun (f): 1 + f(f))
```

Moe Loop?

```
(fun (f): 1 + f(f))(fun (f): 1 + f(f))
```

Moe Loop?

```
(fun (f): 1 + f(f))(fun (f): 1 + f(f))
```

⇒

```
1 + (fun (f): 1 + f(f))(fun (f): 1 + f(f))
```

Moe Loop?

```
1 + (fun (f): 1 + f(f))(fun (f): 1 + f(f))
```

Moe Loop?

```
1 + (fun (f): 1 + f(f))(fun (f): 1 + f(f))
```

⇒

```
1 + 1 + (fun (f): 1 + f(f))(fun (f): 1 + f(f))
```

Tail Calls

```
fun forever(x):
  forever(! x)
```

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

Non-Tail Calls

```
fun run_out_of_memory(x):
  ! 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

```
fun forever(x):
  if x
  | forever(#true)
  | forever(#false)
```

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

```
fun run_out_of_memory(x):
  if run_out_of_memory(x)
  | #true
  | #false
```

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.rhm`:

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

∴ the Shplait continuation is always small

You must maintain this property!