Part I

Defining A Language's Evaluation

`lambda-u.rkt`:

- With **`#lang plai`** ⇒ an eager language
- With **`#lang plai-lazy`** ⇒ a lazy language

Let's take control of evaluation order

Evaluation and "To do" Lists

```
(interp (addC (numC 1) (numC 2)) mt-env)
```

```
⇒ (num+ (interp (numC 1) mt-env)
        (interp (numC 2) mt-env))
```

```
⇒ (interp (numC 1) mt-env)
```

To do:
```
(num+ ●
      (interp (numC 2) mt-env))
```

```
⇒ (numV 1)
```

To do:
```
(num+ ●
      (interp (numC 2) mt-env))
```

```
⇒ (interp (numC 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 (numC 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

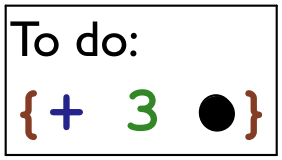
To do:

```
(* ● (f (rest ls)))
(+ 3 ●)
```

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

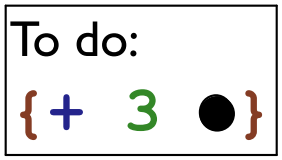
Part 2

Representing Continuations




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

Representing Continuations




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

 (doAddK (numV 3))
```

Representing Continuations

To do:

```
{+ ● {f 0}}
```

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

Representing Continuations

To do:

```
{+ ● {f 0}}
```

```
(define-type Cont
  [addSecondK (r : ExprC) (e : Env)]
  [doAddK (v : Value)]
  ....)

(addSecondK (appC (idC 'f) (numV 0))
            mt-env)
```

Representing Continuations

To do:

```
{+ ● {f 0}}
{+ 3 ●}
```

```
(define-type Cont
  [addSecondK (r : ExprC) (e : Env)]
  [doAddK (v : Value)]
  ....)
```

Representing Continuations

To do:

```
{+ ● {f 0}}
{+ 3 ●}
```

```
(define-type Cont
  [addSecondK (r : ExprC) (e : Env)
              (k : Cont)]
  [doAddK (v : Value)
          (k : Cont)]
  ....)
```

Representing Continuations

To do:

```
{+ ● {f 0}}
{+ 3 ●}
```

```
(define-type Cont
  [doneK]
  [addSecondK (r : ExprC) (e : Env)
              (k : Cont)]
  [doAddK (v : Value)
          (k : Cont)]
  ....)

(addSecondK (appC (idC 'f) (numV 0))
            mt-env
            (doAddK (numV 3)
                    (doneK)))
```

Part 3

`interp` with Continuations

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

`interp` with Continuations

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

`interp` with Continuations

```
(define interp : (ExprC Env -> Value)
  (lambda (a env)
    (type-case ExprC a
      ...
      [addC (l r) (interp l env)
                  (addSecondK r env)]
      ...)))
```

`interp` with Continuations

```
(define interp : (ExprC Env Cont -> Value)
  (lambda (a env k)
    (type-case ExprC a
      ...
      [addC (l r) (interp l env)
                  (addSecondK r env k)]
      ...)))
```

`interp` with Continuations

```
(define interp : (ExprC Env Cont -> Value)
  (lambda (a env k)
    (type-case ExprC a
      ...
      [addC (l r) (interp l env
                          (addSecondK r env k))]
      ...)))
```

`interp` with Continuations

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

`interp` with Continuations

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

`interp` with Continuations

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

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

      ...)))
```

`interp` with Continuations

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

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

`interp` with Continuations

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

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

`interp` with Continuations

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

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

`interp` with Continuations

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

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

Part 4

`interp` with Continuations

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

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

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

⇒ (continue (doneK) (numV 5))

⇒ (numV 5)
```

`interp` with Continuations

```
(define (interp [a : ExprC] [env : Env] [k : Cont]) : Value
  (type-case ExprC a ...
    [addC (l r) (interp l env (addSecondK r env k))]
    ...))

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

```
(interp (addC (numC 5) (numC 2)) mt-env (doneK))

⇒ (interp (numC 5)
          (addSecondK (numC 2) mt-env (doneK)))

⇒ (continue (addSecondK (numC 2) mt-env (doneK))
            (numV 5))
```

`interp` with Continuations

```
(define (interp [a : ExprC] [env : Env] [k : Cont]) : Value
  (type-case ExprC a ...
    [addC (l r) (interp l env (addSecondK r env k))]
    ...))

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

```
⇒ (continue (addSecondK (numC 2) mt-env (doneK))
            (numV 5))

⇒ (interp (numC 2) mt-env
          (doAddK (numV 5) (doneK)))

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

`interp` with Continuations

```
(define (interp [a : ExprC] [env : Env] [k : Cont]) : Value
  (type-case ExprC a ...
    [addC (l r) (interp l env (addSecondK r env k))]
    ...))

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

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

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

`interp` with Continuations

```
(define (interp [a : ExprC] [env : Env] [k : Cont]) : Value
  (type-case ExprC a ...
    [lamC (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 : ExprC] [env : Env] [k : Cont]) : Value
  (type-case ExprC a ...
    [appC (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 ...)])]
    ...))
```

Part 5

Infinite Loop

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

Space-Bounded Loop

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

Book-Language Loop

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

infinite or space-bounded?

Book-Language Loop

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

Book-Language Loop

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

Book-Language Loop

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

Book-Language Loop

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

⇒

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

Book-Language Loop

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

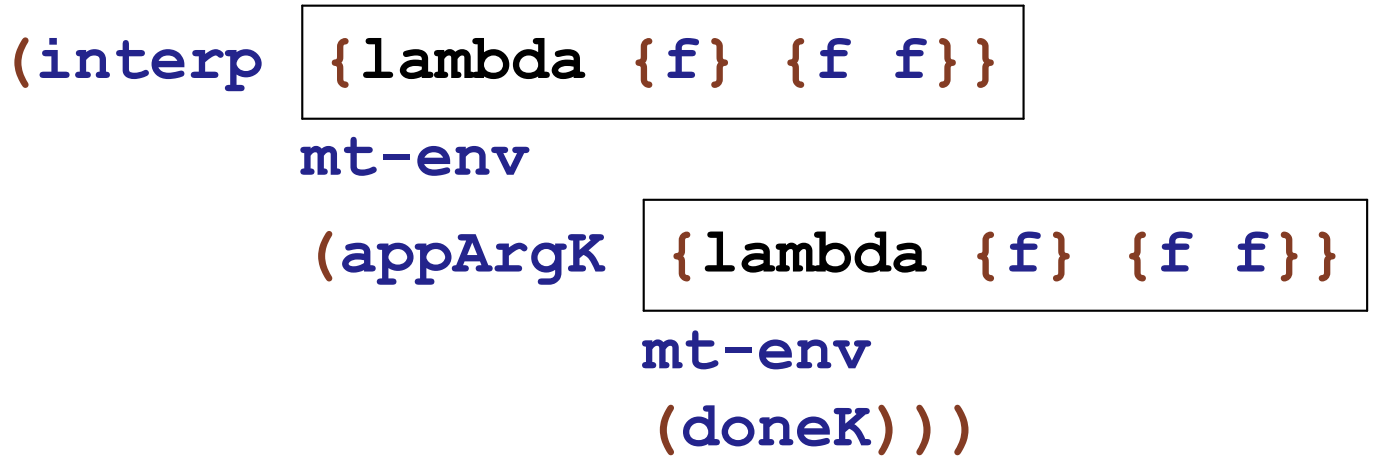
Book-Language Loop

```
(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)))
```

Book-Language Loop



Book-Language 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))
```

Book-Language Loop

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

Book-Language 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)))
```

Book-Language Loop

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

Book-Language 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))
```

Book-Language Loop

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

Book-Language Loop

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

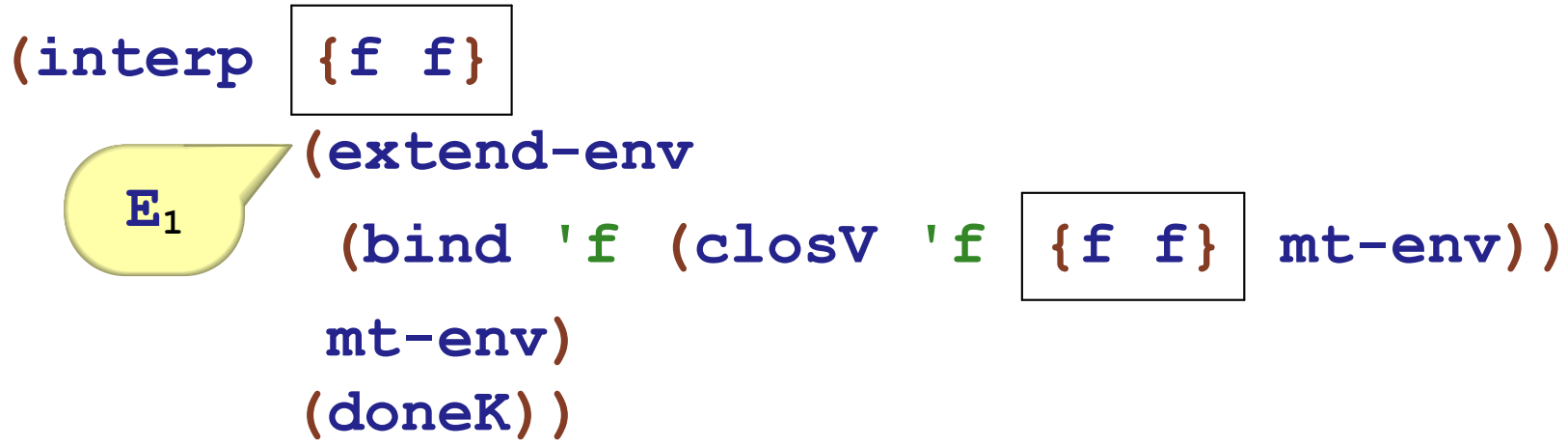
⇒

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

Book-Language Loop

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

Book-Language Loop



Book-Language Loop

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

E_1

=

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

Book-Language Loop

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

Book-Language Loop

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

⇒

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

Book-Language Loop

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

Book-Language Loop

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

⇒

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

Book-Language Loop

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

Book-Language Loop

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

⇒

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

Book-Language Loop

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

Book-Language 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))
```

Book-Language Loop

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

Book-Language Loop

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

⇒

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

Part 6

Book-Language Loop

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

Book-Language Loop

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

Book-Language Loop

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

Book-Language Loop

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

Book-Language Loop

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

Book-Language 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 (zero? 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

```
(define (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.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 **`plai-typed`** continuation is always small