Part 0

Defining Recursion by Expansion

```
letrec name = rhs:
  body
```

could be parsed the same as

```
let name = mk_rec(fun (name): rhs):
  body
```

which is really

```
(fun (name): body)(mk_rec(fun (name): rhs))
```

Part 1

Metacircular Recursion

Recursive Binding

```
block:
  def x = 10
  x + 1
```

```
11
```

Recursive Binding

```
block:
  fun f(x): f(x)
  f(1)
```

infinite loop – good!

Recursive Binding

```
block:
  def x = x
  x
```

x: cannot use before initialization

Recursive Binding

```
block:
  def x = [x]
  x
```

x: cannot use before initialization

Recursive Binding

```
block:
  def f = fun (x): f(x)
  f(1)
```

infinite loop – good!

Recursive Binding

```
letrec f = (fun (x): f(x)):
  f(1)
```

infinite loop – good!

Recursive Binding

```
letrec f = [fun (x): first(f)(x)]:
  first(f)(1)
```

infinite loop – good!

Recursive Binding

```
letrec f = (fun (x): f(x)):
  f(1)
```

```
| letrecE(name, rhs, body):
    block:
      def val = interp(rhs,
                       extend_env(bind(name, val),
                                  env))
      interp(body,
             extend_env(bind(name, val),
                        env))
```

val: cannot use before initialization

Recursive Binding

```
letrec f = (fun (x): f(x)):
  f(1)
```

```
| letrecE(name, rhs, body):
    block:
      def val = interp(rhs,
                       extend_env(bind(name, fun (): val),
                                  env))
      interp(body,
             extend_env(bind(name, fun (): val),
                        env))
```

works!

Metacircular `letrec`

```
type Binding
| bind(name :: Symbol,
       val :: () -> Value)

fun lookup(n :: Symbol, env :: Env) :: Value:
  match env
  | []: error(#'lookup, "free variable")
  | cons(b, rst_env): cond
                      | n == bind.name(b):
                          bind.<val(b)()
                      | ~else: lookup(n, rst_env)
```

Part 2

Exp Grammar

```
<Exp>  ::=  <Int>
         |  <Exp> + <Exp>
         |  <Exp> - <Exp>
         |  <Symbol>
         |  fun (<Symbol>): <Exp>
         |  <Exp>(<Exp>)
         |  let <Symbol> = <Exp>: <Exp>
         |  if <Exp> == 0 | <Exp> | <Exp>            NEW
         |  letrec <Symbol> = <Exp>: <Exp>           NEW
```

Metacircular `letrec`

```
fun interp(a :: Exp, env :: Env) :: Value:
  match a
  | ....
  | letrecE(n, rhs, body):
      def new_env:
        extend_env(bind(n, fun (): val),
                   env)
      def val:
        interp(rhs, new_env)
      interp(body, new_env)
```

Metacircular `letrec`

```
check:
  interp(parse('letrec fac = (fun (x):
                                      if x == 0
                                      | 1
                                      | x * fac(x + -1)):
                  fac(5)'),
         mt_env)
  ~is intV(120)
```

Metacircular `letrec`

```
interp(parse('letrec x = x:
                x'),
       mt_env)
```

val: cannot use before initialization

... a crash at the Shplait level

Part 3

Assignment-Based Recursion

Defining Recursion by Expansion

```
letrec name = rhs:
  body
```

could be parsed the same as

```
let name = mk_rec(fun (name): rhs):
  body
```

which is really

```
(fun (name): body)(mk_rec(fun (name): rhs))
```

Defining Recursion by Expansion

Another approach:

```
letrec fac = (fun (n):
                if n == 0
                | 1
                | n * fac(n - 1)):
  fac(10)
```

⇒

```
let fac = 42:
  begin:
    fac := (fun (n):
              if n == 0
              | 1
              | n * fac(n - 1))
    fac(10)
```

Implementing Recursion

Expanding to assignment in Moe works only if Moe has state...

... but the same state idea for `letrec` can work using Shplait's state

Assignment-Based `letrec`

```
type Binding
| bind(name :: Symbol,
       val :: Boxof(Value))

fun lookup(n :: Symbol, env :: Env) :: Value:
  match env
  | []: error(#'lookup, "free variable")
  | cons(b, rst_env): cond
                      | n == bind.name(b):
                          unbox(bind.val(b))
                      | ~else: lookup(n, rst_env)
```

Assignment-Based `letrec`

```
fun interp(a :: Exp, env :: Env) :: Value:
  match a
  | ....
  | letrecE(n, rhs, body):
      let b = box(intV(42)):
        let new_env = extend_env(bind(n, b),
                                 env):
          set_box(b, interp(rhs, new_env))
          interp(body, new_env)
```

Part 4

Use Before Initialization

Use Before Initialization

```
interp(parse('letrec x = x:
                x'),
       mt_env)
```

⇒ `intV(42)`

Use Before Initialization

```
fun interp(a :: Exp, env :: Env) :: Value:
  match a
  | ....
  | letrecE(n, rhs, body):
      let b = box(intV(42)):
        let new_env = extend_env(bind(n, b),
                                 env):
          set_box(b, interp(rhs, new_env))
          interp(body, new_env)
```

Use Before Initialization

```
type Binding
| bind(name :: Symbol,
       val :: Boxof(Optionof(Value)))

fun lookup(n :: Symbol, env :: Env) :: Value:
  match env
  | []: error(#'lookup, "free variable")
  | cons(b, rst_env):
      cond
      | n == bind.name(b):
          match unbox(bind.val(b))
          | none(): error(#'lookup, "use before initialization")
          | some(v): v
      | ~else: lookup(n, rst_env)
```

Use Before Initialization

```
fun interp(a :: Exp, env :: Env) :: Value:
  match a
  | ....
  | letrecE(n, rhs, body):
      let b = box(none()):
        let new_env = extend_env(bind(n, b),
                                 env):
          set_box(b, some(interp(rhs, new_env)))
          interp(body, new_env)
```