Part 0

Defining Recursion by Expansion

```
{letrec {[name rhs]}
  body}
```

could be parsed the same as

```
{let {[name {mk-rec {lambda {name} rhs}}]}
  body}
```

which is really

```
{{lambda {name} body}
 {mk-rec {lambda {name} rhs}}}
```

Part 1

Metacircular Recursion

Recursive Binding

```
(local [(define x 10)]
  (+ x 1))
```

```
11
```

Recursive Binding

```
(local [(define (f x)
          (f x))]
  (f 1))
```

infinite loop

Recursive Binding

```
(local [(define x x)]
  x)
```

```
#<undefined>
```

Recursive Binding

```
(local [(define x (list x))]
  x)
```

```
(list #<undefined>)
```

Recursive Binding

```
(local [(define f
          (lambda (x) (f x)))]
  (f 1))
```

infinite loop

Recursive Binding

```
(letrec ([f (lambda (x) (f x))])
  (f 1))
```

infinite loop

Recursive Binding

```
(letrec ([f
          (list
           (lambda (x) ((first f) x)))])
  ((first f) 1))
```

infinite loop

Recursive Binding

```
{letrec {[x 10]}
  {+ x 1}}
```

```
(letrec ([val
          (interp (numC 10)
                  (extend-env (bind 'x val)
                              env))])
  (interp (plusC (idC 'x) (numC 1))
          (extend-env (bind 'x val)
                      env)))
```

seems to work...

Recursive Binding

```
{letrec {[f {lambda {x}
              {f x}}]}
  {f 1}}
```

```
(letrec ([val
          (interp (lamC 'x (appC (idC 'f)
                                 (idC 'x)))
                  (extend-env (bind 'f val)
                              env))])
  (interp (appC (idC 'f) (numC 1))
          (extend-env (bind 'f val)
                      env)))
```

contract failure

Recursive Binding

```
{letrec {[f {lambda {x}
                    {f x}}]}
  {f 1}}
```

```
(letrec ([new-env (extend-env (bind 'f (lambda ()
                                         val))
                              env)]
         [val (interp (lamC 'x (appC (idC 'f)
                                     (idC 'x)))
                      new-env)])
  (interp (appC (idC 'f) (numC 1)) new-env))
```

works!

Metacircular `letrec`

```
(define-type Binding
  [bind (name : symbol)
        (val : (-> Value))])

(define (lookup [n : symbol] [env : Env]) : Value
  (cond
   [(empty? env) (error 'lookup "free variable")]
   [else (if (symbol=? n (bind-name (first env)))
             ((bind-val (first env)))
             (lookup n (rest env)))]))
```

Part 2

Expr Grammar

```
<Expr>  ::=  <Num>
          |  {+ <Expr> <Expr>}
          |  {- <Expr> <Expr>}
          |  <Sym>
          |  {lambda {<Sym>} <Expr>}
          |  {<Expr> <Expr>}
          |  {let {[<Sym> <Expr>]} <Expr>}
          |  {if0 <Expr> <Expr> <Expr>}          NEW
          |  {letrec {[<Sym> <Expr>]} <Expr>}    NEW
```

Metacircular `letrec`

```
(define (interp [a : Expr] [env : Env]) : Value
  (type-case Expr a
    ....
    [letrecC (n rhs body)
             (letrec ([new-env
                       (extend-env
                        (bind n (lambda () val))
                        env)]
                      [val
                       (interp rhs new-env)])
               (interp body new-env))]))
```

Part 3

Assignment-Based Recursion

Defining Recursion by Expansion

```
{letrec {[name rhs]}
  body}
```

could be parsed the same as

```
{let {[name {mk-rec {lambda {name} rhs}}]}
  body}
```

which is really

```
{{lambda {name} body}
 {mk-rec {lambda {name} rhs}}}
```

Defining Recursion by Expansion

Another approach:

```
(letrec ([fac
          (lambda (n)
            (if (zero? n)
                1
                (* n (fac (- n 1)))))])
  (fac 10))
```

⇒

```
(let ([fac 42])
  (begin
    (set! fac
          (lambda (n)
            (if (zero? n)
                1
                (* n (fac (- n 1))))))
    (fac 10)))
```

Implementing Recursion

The **`set!`** approach to definition works only when the defined language includes **`set!`**

But the **`set!`** approach to implementation requires only that the implementation language includes **`set!`**...

Assignment-Based `letrec`

```
(define-type Binding
  [bind (name : symbol)
        (val : (boxof Value))])

(define (lookup [n : symbol] [env : Env]) : Value
  (cond
   [(empty? env) (error 'lookup "free variable")]
   [else (if (symbol=? n (bind-name (first env)))
             (unbox (bind-val (first env)))
             (lookup n (rest env)))]))
```

Assignment-Based `letrec`

```
(define (interp [a : Expr] [env : Env]) : Value
  (type-case Expr a
    ....
    [letrecC (n rhs body)
     (let ([b (box (numV 42))])
       (let ([new-env (extend-env
                       (bind n b)
                       env)])
         (begin
           (set-box! b (interp rhs new-env))
           (interp body new-env))))]))
```

Part 4

Cyclic Data

Cycles

```
(letrec ([f (lambda (x)
              (f x))])
  ...)
```

Cycles