Part I

Meta-Circular Recursion
Recursive Binding

{\texttt{rec \{x x\} x}}

a closure (!)
Recursive Binding

\{\{\text{rec} \ \{x \ x\} \ x\} \ 10\}

infinite loop (!)
Recursive Binding

{with {f {fun {g} {g g}}} {f f}}

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 x) (f x))]
  (f 1))
```

infinite loop
Recursive Binding

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

infinite loop
Recursive Binding

\[
\text{(local [\(\text{define f}\)}
  \text{(list}
    \text{(lambda (x) ((first f) x))))]
\text{(first f) 1))}
\]

infinite loop
Recursive Binding

```lisp
(local [(define val
          (interp (num 10)
              (aSub 'x
                  val
                  ds))))]

val)

contract failure
```
Recursive Binding

```
(local [(define val
  (interp (num 10)
    (aSub 'x
      (lambda () val)
    ds)))]
  val)
```

could work
Recursive Binding

(local [(define new-ds
         (aSub 'x
             (lambda () val)
             ds))
       (define val
         (interp (num 10)
                 new-ds))]
      (interp (id 'x) new-ds))

could work
(define-type DefrdSub
  [mtSub]
  [aSub (name symbol?)
    (get-value (-> FAE-Value?)
    (rest DefrdSub?)]])

(define (lookup name ds)
  (type-case DefrdSub ds
    [mtSub () (error 'lookup "free variable")]
    [aSub (sub-name get-num rest-ds)
      (if (symbol=? sub-name name)
        (get-num)
        (lookup name rest-ds)]]))
Metacircular Recursion

(define-type FAE

....
[rec (name symbol?)
   (named-expr FAE?)
   (body FAE?)])

(define (interp a-fae ds)
  (type-case FAE a-fae

....
[rec (name named-expr body-expr)
   (local [(define new-ds
      (aSub name
         (lambda () val)
         ds))
      (define val (interp named-expr
data new-ds)])
      (interp body-expr new-ds))]]))
Part II

Assignment-Based Recursion
Defining Recursion by Expansion

\[
\text{rec } \{ \langle \text{id}\rangle_1 \langle \text{FAE}\rangle_1 \}
\]

\[
\langle \text{FAE}\rangle_2 \}
\]

could be parsed the same as

\[
\text{with } \{ \langle \text{id}\rangle_1 \langle mk-rec \langle \text{id}\rangle_1 \langle \text{FAE}\rangle_1 \} \}
\]

\[
\langle \text{FAE}\rangle_2 \}
\]

which is really

\[
\{ \{ \text{fun } \langle \text{id}\rangle_1 \langle \text{FAE}\rangle_2 \}
\]

\[
\langle mk-rec \langle \text{fun } \langle \text{id}\rangle_1 \langle \text{FAE}\rangle_1 \} \}
\]
Defining Recursion by Expansion

Another approach:

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

⇒

```
(let ([fac 42])
  (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!**...
RCFAE Grammar

<RCFAE> ::= <num>
| {+ <RCFAE> <RCFAE>}
| {- <RCFAE> <RCFAE>}
| <id>
| {fun {<id>} <RCFAE>}
| {<RCFAE> <RCFAE>}
| {if0 <RCFAE> <RCFAE> <RCFAE>}
| {rec {<id> <RCFAE>} <RCFAE>}

NEW
NEW
RCFAE Datatype

(define-type RCFAE
  [num (n number?)])
[add (lhs RCFAE?)
  (rhs RCFAE?)]
[sub (lhs RCFAE?)
  (rhs RCFAE?)]
[id (name symbol?)]
[fun (param symbol?)
  (body RCFAE?)]
[app (fun-expr RCFAE?)
  (arg-expr RCFAE?)]
[if0 (test-expr RCFAE?)
  (then-expr RCFAE?)
  (else-expr RCFAE?)]
[rec (name symbol?)
  (named-expr RCFAE?)
  (body RCFAE?)]]
RCFAE Interpreter

; interp : RCFAE DefrdSub -> RCFAE-Value
(define (interp a-rcf ae ds)
  (type-case RCFAE a-rcf ae
    [num (n) (numV n)]
    [add (l r) (num+ (interp l ds) (interp r ds))]
    [sub (l r) (num- (interp l ds) (interp r ds))]
    [id (name) (lookup name ds)]
    [fun (param body-expr)
      (closureV param body-expr ds)]
    [app (fun-expr arg-expr)
      (local [(define fun-val
               (interp fun-expr ds))]
               (interp (closureV-body fun-val)
               (aSub (closureV-param fun-val)
               (interp arg-expr ds)
               (closureV-sc fun-val))))]]

[if0 (test-expr then-expr else-expr)
  ...]
[rec (bound-id named-expr body-expr)
  ...])}
RCFAE Interpreter

; interp : RCFAE DefrdSub -> RCFAE-Value
(define (interp a-rcfae ds)
  (type-case RCFAE a-rcfae
    [num (n) (numV n)]
    [add (l r) (num+ (interp l ds) (interp r ds))]
    [sub (l r) (num- (interp l ds) (interp r ds))]
    [id (name) (lookup name ds)]
    [fun (param body-expr)
      (closureV param body-expr ds)]
    [app (fun-expr arg-expr)
      (local [(define fun-val
                (interp fun-expr ds))]
          (interp (closureV-body fun-val)
              (aSub (closureV-param fun-val)
                  (interp arg-expr ds)
                  (closureV-sc fun-val))))]
    [if0 (test-expr then-expr else-expr)
      ... (interp test-expr ds)
      ... (interp then-expr ds)
      ... (interp else-expr ds) ...]
    [rec (bound-id named-expr body-expr)
      ...])]
RCFAE Interpreter

; interp : RCFAE DefrdSub -> RCFAE-Value
(define (interp a-rcfae ds)
  (type-case RCFAE a-rcfae
    [num (n) (numV n)]
    [add (l r) (num+ (interp l ds) (interp r ds))]
    [sub (l r) (num- (interp l ds) (interp r ds))]
    [id (name) (lookup name ds)]
    [fun (param body-expr)
      (closureV param body-expr ds)]
    [app (fun-expr arg-expr)
      (local [(define fun-val
        (interp fun-expr ds))]
        (interp (closureV-body fun-val)
          (aSub (closureV-param fun-val)
            (interp arg-expr ds)
              (closureV-sc fun-val))))]
    [if0 (test-expr then-expr else-expr)
      (if (numzero? (interp test-expr ds))
        (interp then-expr ds)
          (interp else-expr ds))]
    [rec (bound-id named-expr body-expr)
      ...])))
Testing For Zero

; numzero? : RCFAE-Value -> boolean
(define (numzero? n)
  (zero? (numV-n n)))
; interp : RCFAE DefrdSub -> RCFAE-Value
(define (interp a-rcfae ds)
  (type-case RCFAE a-rcfae
    ...
    [rec (bound-id named-expr body-expr)
      (local [(define value-holder (box (numV 42)))
        (define new-ds (aRecSub bound-id
                             value-holder
                             ds))]
        (begin
          (set-box! value-holder (interp named-expr new-ds))
          (interp body-expr new-ds)))])))
RCFAE DefrdSub

(define-type DefrdSub
    [mtSub]
    [aSub (name symbol?)
        (value RCFAE-Value?)
        (sc DefrdSub?)]
    [aRecSub (name symbol?)
        (value-box (box-of RCFAE-Value?)
        (sc DefrdSub?)])

(define-type RCFAE-Value
    [numV (n number?)]
    [closureV (param symbol?)
        (body RCFAE?)
        (sc DefrdSub?)])

(define (box-of pred)
    (lambda (x)
        (and (box? x) (pred (unbox x))))))
; lookup : symbol DefrdSub -> num
(define (lookup name ds)
  (type-case DefrdSub ds
    [mtSub () (error 'lookup "free variable")]
    [aSub (sub-name val rest-sc)
      (if (symbol=? sub-name name)
        val
        (lookup name rest-sc))]
    [aRecSub (sub-name val-box rest-sc)
      (if (symbol=? sub-name name)
        (unbox val-box)
        (lookup name rest-sc)]]))