Recursion as a Built-In
Recursion, this time in the language

\[
\{ \text{rec } \{ \text{id}_1 \text{ } <\text{FAE}_1 \} \\
\text{<FAE>_2} \}
\]

like \text{with} but \text{id}_1 is bound in \text{<FAE>_2} \text{ and } \text{<FAE>_1}
Defining Recursion

Last time:

\[
\{ \text{rec } \{ <\text{id}>_1 \ <\text{FAE}>_1 \} \\
<\text{FAE}>_2 \}
\]

could be parsed the same as

\[
\{ \text{with } \{ \text{mk-rec } \ldots \text{mk-rec-code} \ldots \} \\
\{ \text{with } \{ <\text{id}>_1 \ \{ \text{mk-rec } \{ \text{fun } \{ <\text{id}>_1 \} \\
<\text{FAE}>_1 \} \} \} \} \\
<\text{FAE}>_2 \}
\]
Defining Recursion

which is really

```haskell
{{fun {mk-rec}

  {{fun {<id>_1} <FAE>_2}
  {mk-rec {fun {<id>_1}
    <FAE>_1}}}}}

...mk-rec-code...
```
Defining Recursion

Another approach:

```scheme
(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))
```
Defining Recursion

With explicit data structure mutation:

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

⇒

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

⇒ 6
Implementing Recursion

The \texttt{set!} approach to \textit{definition} works only when the object language includes \texttt{set!}.
  - I.e., for programs in our object language to use that trick, our object language needs \texttt{set!}.

But the \texttt{set!} approach to \textit{implementation} requires only that the meta language includes \texttt{set}!...
  - I.e., for our interpreter to use that trick, our meta language needs \texttt{set}!.
RCFAE Grammar

\[ <\text{RCFAE}> ::= <\text{num}> \]

\[ \mid \{ + <\text{RCFAE}> <\text{RCFAE}>\} \]

\[ \mid \{ - <\text{RCFAE}> <\text{RCFAE}>\} \]

\[ \mid <\text{id}> \]

\[ \mid \{\text{fun} \{ <\text{id}>\} <\text{RCFAE}>\} \]

\[ \mid \{ <\text{RCFAE}> <\text{RCFAE}>\} \]

\[ \mid \{\text{if0} <\text{RCFAE}> <\text{RCFAE}> <\text{RCFAE}>\} \]

\[ \mid \{\text{rec} \{ <\text{id}> <\text{RCFAE}>\} <\text{RCFAE}>\} \]
(define-type RCFAE
  [num (n number?)])
[add (lhs RCFAE?)
  (rhs RCFAE?)]
[sub (lhs RCFAE?)
  (rhs RCFAE?)]
[id (name symbol?)]
[fun (param-name 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? DefSub? -> 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-name body)
      (closureV param-name body ds)]
    [app (fun-expr arg-expr)
      (local [(define fun-val
        (interp fun-expr ds))]
        (interp (closureV-body fun-val)
          (aSub (closureV-param-name fun-val)
            (interp arg-expr ds)
            (closureV-ds fun-val))))]
    [if0 (test-expr then-expr else-expr)
      ...]
    [rec (name named-expr body)
      ...]))
RCFAE Interpreter

; interp : RCFAE? DefSub? -> 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-name body)
        (closureV param-name body ds)]
    [app (fun-expr arg-expr)
        (local [(define fun-val
            (interp fun-expr ds))]
            (interp (closureV-body fun-val)
                (aSub (closureV-param-name name fun-val)
                    (interp arg expr ds)
                    (closureV-ds fun-val))))])
    [if0 (test-expr then-expr else-expr)
      ... (interp test-expr ds)
      ... (interp then-expr ds)
      ... (interp else-expr ds) ...]
    [rec (name named-expr body)
      ...]))
RCFAE Interpreter

; interp : RCFAE? DefSub? -> 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-name body)
     (closureV param-name body ds)]
   [app (fun-expr arg-expr)
     (local [(define fun-val
               (interp fun-expr ds))]
     (interp (closureV-body fun-val)
       (aSub (closureV-param-name param-name fun-val)
         (interp arg-expr ds)
         (closureV-ds fun-val)))]
   [if0 (test-expr then-expr else-expr)
     (if (zero? (numV-n (interp test-expr ds)))
       (interp then-expr ds)
       (interp else-expr ds))]
   [rec (name named-expr body)
     ...]))
RCFAE Interpreter

; interp : RCFAE? DefSub? -> RCFAE-Value?
(define (interp a-rcfae ds)
  (type-case RCFAE a-rcfae
    ...
    [rec (name named-expr body)
      (local [(define value-holder (box (numV 42)))
        (define new-ds (aRecSub name
                         value-holder
                         ds))]
        (set-box! value-holder (interp named-expr new-ds))
        (interp body new-ds))])))
(define-type DefSub
  [mtSub]
  [aSub (name symbol?)
    (value RCFAE-Value?)
    (ds DefSub?)]]
  [aRecSub (name symbol?)
    (value-box (box/c RCFAE-Value?))
    (ds DefSub?)])

(define-type RCFAE-Value
  [numV (n number?)])
  [closureV (param-name symbol?)
    (body RCFAE?)
    (ds DefSub?)])
RCFAE Lookup

; lookup : symbol? DefSub? -> RCFAE-Value?
(define (lookup name ds)
  (type-case DefSub ds
    [mtSub () (error 'lookup "free variable")]
    [aSub (n val rest)
      (if (symbol=? n name)
          val
          (lookup name rest))]
    [aRecSub (n val-box rest)
      (if (symbol=? n name)
          (unbox val-box)
          (lookup name rest))])))