Recursion, this time in the language

\{\texttt{rec} \{<\texttt{id}>_1 <\texttt{FAE}>_1\}\}

\texttt{<FAE>_2}\}

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

Last time:

\[
\begin{align*}
{\text{rec}} & \{ <\text{id}>_1, <\text{FAE}>_1 \} \\
& \{ <\text{FAE}>_2 \}
\end{align*}
\]

could be parsed the same as

\[
\begin{align*}
{\text{with}} & \{ \{ \text{mk-rec} \ldots \text{mk-rec-code}\ldots \} \\
& \{ \text{with} \{ <\text{id}>_1, \{ \text{mk-rec} \{ \text{fun} \{ <\text{id}>_1 \} \} \} \} \} \\
& \{ <\text{FAE}>_2 \} \}
\end{align*}
\]
Defining Recursion

which is really

\[
\begin{align*}
&\{\{\text{fun} \{\text{mk-rec}\}\} \\
&\quad \{\{\text{fun} \{<\text{id}>_1\} <\text{FAE}>_2\} \\
&\quad \{\text{mk-rec} \{\text{fun} \{<\text{id}>_1\} \\
&\quad \quad <\text{FAE}>_1\}\}\}\}
\end{align*}
\]

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

Another approach:

\[
\begin{align*}
\text{(local } & \text{[(define fac} \\
& \text{\ (lambda } (n) \\
& \text{\ (if } (\text{zero? } n) \\
& \text{\ 1} \\
& \text{\ (* } n \text{\ (fac (- } n \text{ 1))))))))] \\
\text{(fac 10)})
\end{align*}
\]

⇒

\[
\begin{align*}
\text{(let } & \text{[(fac 42)]} \\
& \text{(set! fac} \\
& \text{\ (lambda } (n) \\
& \text{\ (if } (\text{zero? } n) \\
& \text{\ 1} \\
& \text{\ (* } n \text{\ (fac (- } n \text{ 1))))))))) \\
\text{(fac 10)})
\end{align*}
\]
Implementing Recursion

The \texttt{set!} approach to definition works only when the defined language includes \texttt{set!}.

But the \texttt{set!} approach to implementation requires only that the implementation language includes \texttt{set!}...
RCFAE Grammar

\[ \text{<RCFAE>} ::= \text{<num>} \]
\[ \quad | \quad \{+ \text{<RCFAE>} \text{<RCFAE>}\} \]
\[ \quad | \quad \{- \text{<RCFAE>} \text{<RCFAE>}\} \]
\[ \quad | \quad \text{<id>} \]
\[ \quad | \quad \{\text{fun} \{\text{id}\} \text{<RCFAE>}\} \]
\[ \quad | \quad \{\text{<RCFAE>} \text{<RCFAE>}\} \]
\[ \quad | \quad \{\text{if0} \text{<RCFAE>} \text{<RCFAE>} \text{<RCFAE>}\} \]
\[ \quad | \quad \{\text{rec} \{\text{id} \text{<RCFAE>}\} \text{<RCFAE>}\} \]
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-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)
      ...]
    [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-exparg arg-exparg)
    (local [(define fun-val
      (interp fun-exparg ds))]
      (interp (closureV-body fun-val)
        (aSub (closureV-param fun-val
          (interp arg-exparg ds)
            (closureV-sc fun-val)))))]
  [if0 (test-exp expr then-exp expr else-exp expr)
    (if (numzero? (interp test-exp expr ds))
      (interp then-exp expr ds)
      (interp else-exp expr ds))]
  [rec (bound-id named-exp body-exp)
    ...]])
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)
      ...]]))
RCFAE Interpreter

; interp : RCFAE DefrdSub -> RCFAE-Value
(define (interp a-rcfae ds)
  (type-case RCFAE a-rcfae
    ...
    [rec (bound-id named-expr body-expr)
      ... (interp named-expr ds)
      ... (interp body-expr ds) ...]])
RCFAE Interpreter

; interp : RCFAE DefrdSub -> RCFAE-Value
(define (interp a-rcfae ds)
  (type-case RCFAE a-rcfae
      ...
      [rec (bound-id named-expr body-expr)
          (local [(define new-ds (aRecSub bound-id
              ...
              ds)])
            ...
            (interp named-expr new-ds)
            ...
            (interp body-expr new-ds) ...)])})
RCFAE Interpreter

; 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))]
      ... (interp named-expr new-ds)
      ... (interp body-expr new-ds) ...)])))
RCFAE Interpreter

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

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

; 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))])))