Defining Recursion

Last time:

\[
\text{rec } \{ \langle \text{id} \rangle_1 \ <\text{FAE}\rangle_1 \}
\]
\[
\quad <\text{FAE}\rangle_2
\]

could be parsed the same as

\[
\text{with } \{ \langle \text{id} \rangle_1 \ \text{mk-rec } \{ \text{fun } \{ \langle \text{id} \rangle_1 \ <\text{FAE}\rangle_1 \}\}\}
\]
\[
\quad <\text{FAE}\rangle_2
\]

which is really

\[
\{\{ \text{fun } \{ \langle \text{id} \rangle_1 \ <\text{FAE}\rangle_2 \}
\quad \text{mk-rec } \{ \text{fun } \{ \langle \text{id} \rangle_1 \ <\text{FAE}\rangle_1 \}\}\}\}
\]
Defining Recursion

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

; 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) ...)))
; 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)]]))