Defining Recursion

Last time:

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

could be parsed the same as

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

which is really

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

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

\[
| \quad \{+ \text{<RCFAE>} \text{<RCFAE>}\}
\]

\[
| \quad \{- \text{<RCFAE>} \text{<RCFAE>}\}
\]

\[
| \quad \text{<id>}
\]

\[
| \quad \{\text{fun} \{\text{id}\} \text{<RCFAE>}\}
\]

\[
| \quad \{\text{<RCFAE>} \text{<RCFAE>}\}
\]

\[
| \quad \{\text{if0} \text{<RCFAE>} \text{<RCFAE>} \text{<RCFAE>}\}
\]

\[
| \quad \{\text{rec} \{\text{id} \text{<RCFAE>}\} \text{<RCFAE>}\}
\]

New
(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-exp expr arg-expr)
      (local [(define fun-val
                    (interp fun-exp ds))]
        (interp (closureV-body fun-val)
          (aSub (closureV-param fun-val)
            (interp arg-expr ds)
            (closureV-sc fun-val))))]
    [if0 (test-exp then-exp else-exp)
      ... (interp test-exp ds)
      ... (interp then-exp ds)
      ... (interp else-exp ds) ...]
    [rec (bound-id named-exp 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-exp) (closureV param body-exp ds)]
    [app (fun-exp arg-exp) (local [(define fun-val
      (interp fun-exp ds))]
      (interp (closureV-body fun-val)
        (aSub (closureV-param fun-val)
          (interp arg-exp ds)
          (closureV-sc fun-val)))]
    [if0 (test-exp then-exp else-exp)
      (if (numzero? (interp test-exp ds))
        (interp then-exp ds)
        (interp else-exp 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)
      (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-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))))))
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))])