## Defining Recursion

Last time:

```
{rec {<id>1 <FAE>1}
  <FAE>2}
```

could be parsed the same as

```
{with {<id>1 {mk-rec {fun {<id>1} <FAE>1}}}
  <FAE>2}
```

which is really

```
{{fun {<id>1} <FAE>2}
 {mk-rec {fun {<id>1} <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 **`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>}     NEW
           |  {rec {<id> <RCFAE>} <RCFAE>}      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 SubCache -> RCFAE-Value
(define (interp a-rcfae sc)
  (type-case RCFAE a-rcfae
    [num (n) (numV n)]
    [add (l r) (num+ (interp l sc) (interp r sc))]
    [sub (l r) (num- (interp l sc) (interp r sc))]
    [id (name) (lookup name sc)]
    [fun (param body-expr)
         (closureV param body-expr sc)]
    [app (fun-expr arg-expr)
         (local [(define fun-val
                   (interp fun-expr sc))]
           (interp (closureV-body fun-val)
                   (aSub (closureV-param fun-val)
                         (interp arg-expr sc)
                         (closureV-sc fun-val)))))]
    [if0 (test-expr then-expr else-expr)
         ...]
    [rec (bound-id named-expr body-expr)
      ...]))
```

# RCFAE Interpreter

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

## RCFAE Interpreter

```
; interp : RCFAE SubCache -> RCFAE-Value
(define (interp a-rcfae sc)
  (type-case RCFAE a-rcfae
    [num (n) (numV n)]
    [add (l r) (num+ (interp l sc) (interp r sc))]
    [sub (l r) (num- (interp l sc) (interp r sc))]
    [id (name) (lookup name sc)]
    [fun (param body-expr)
         (closureV param body-expr sc)]
    [app (fun-expr arg-expr)
         (local [(define fun-val
                   (interp fun-expr sc))]
           (interp (closureV-body fun-val)
                   (aSub (closureV-param fun-val)
                         (interp arg-expr sc)
                         (closureV-sc fun-val)))))]
    [if0 (test-expr then-expr else-expr)
         (if (numzero? (interp test-expr sc))
             (interp then-expr sc)
             (interp else-expr sc))]
    [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 SubCache -> RCFAE-Value
(define (interp a-rcfae sc)
  (type-case RCFAE a-rcfae
    ...
    [rec (bound-id named-expr body-expr)
      (local [(define value-holder (box (numV 42)))
              (define new-sc (aRecSub bound-id
                                      value-holder
                                      sc))]
        (begin
          (set-box! value-holder (interp named-expr new-sc))
          (interp body-expr new-sc)))]))
```

## RCFAE SubCache

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

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

(define (box-of pred)
  (lambda (x)
    (and (box? x) (pred (unbox x)))))
```

## RCFAE Lookup

```
; lookup : symbol SubCache -> num
(define (lookup name sc)
  (type-case SubCache sc
    [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))]))
```