Scheme vs. Algebra

In Scheme, we have a specific order for evaluating sub-expressions:

`(+ (* 4 3) (- 8 7))` $\Rightarrow$ `(+ 12 (- 8 7))` $\Rightarrow$ `(+ 12 1)`

In Algebra, order doesn’t matter:

$$(4\cdot3)+(8-7) \Rightarrow 12+(8-7) \Rightarrow 12+1$$

or

$$(4\cdot3)+(8-7) \Rightarrow (4\cdot3)+1 \Rightarrow 12+1$$

Algebraic Shortcuts

In Algebra, if we see

f(x, y) = x

g(z) = ...

f(17, g(g(g(g(g(18))))))

then we can go straight to

17

because the result of all the g calls will not be used

But why would a programmer write something like that?

Avoiding Unnecessary Work

```
; layout-text : string w h -> pict
(define (layout-text txt w h)
  (local [(define lines
            ; lots of work to flow a paragraph
            ...)]
    (make-pict w
               h
               (lambda (dc x y)
                 ; draw paragraph lines
                 ...))))
...
(define speech (layout-text "Four score..."
                            800
                            600))
...
(pict-width speech)
```

Avoiding Unnecessary Work

```
; read-all-chars : file -> list-of-char
(define (read-all-chars f)
  (if (at-eof? f)
      empty
      (cons (read-char f) (read-all-chars f))))
...
(define content (read-all-chars (open-file user-file)))
(if (equal? (first content) #\#)
    (process-file (rest content))
    (error 'parser "not a valid file"))
```

Recursive Definitions

```
; numbers-from : int -> list-of-int
(define (numbers-from n)
  (cons n (numbers-from (add1 n))))
...
(define nonneg (numbers-from 0))
(list-ref nonneg 10675)
```

Lazy Evaluation

Languages like Scheme, Java, and C are called ***eager***

- An expression is evaluated when it is encountered

Languages that avoid unnecessary work are called ***lazy***

- An expression is evaluated only if its result is needed

Lazy Evaluation in DrScheme

`plai-lazy.plt` adds a **PLAI Lazy** language to DrScheme

In the **Choose Language..** dialog, click
Show Details and then
Syntactic test suite coverage

(Works for both eager and lazy languages)

- Green means evaluated at least once
- Red means not yet evaluated
- Normal coloring is the same as all green

Interepreter in Lazy Scheme

Doesn’t work because result of **`set-box!`** is never used:

```
(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)))]))
```

Interepreter in Lazy Scheme

Working implementation is actually simpler:

```
(define (interp a-rcfae sc)
  (type-case RCFAE a-rcfae
    ...
    [rec (bound-id named-expr body-expr)
      (local [(define new-sc (aSub bound-id
                                   (interp named-expr new-sc)
                                   sc))]
        (interp body-expr new-sc))]))
```