Racket vs. Algebra

In Racket, 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:

\[(4 \cdot 3) + (8 - 7) \Rightarrow 12 + (8 - 7) \Rightarrow 12 + 1\]

or

\[(4 \cdot 3) + (8 - 7) \Rightarrow (4 \cdot 3) + 1 \Rightarrow 12 + 1\]
Algebraic Shortcuts

In Algebra, if we see

\[ f(x, y) = x \]

\[ g(z) = \ldots \]

\[ f(17, g(g(g(g(18)))))) \]

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 Racket, 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 DrRacket

`plai-lazy.plt` adds a **PLAI Lazy** language to DrRacket: `#lang plai-lazy`

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
RCFAE Interpreter in Lazy Racket

 Doesn’t work because result of \texttt{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)))]))
RCFAE Interpreter in Lazy Racket

Working implementation is actually simpler:

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

\[
\langle \text{CFAL} \rangle \ ::= \ \langle \text{num} \rangle \\
| \ \{ + \ \langle \text{CFAL} \rangle \ \langle \text{CFAL} \rangle \} \\
| \ \{ - \ \langle \text{CFAL} \rangle \ \langle \text{CFAL} \rangle \} \\
| \ \langle \text{id} \rangle \\
| \ \{ \text{fun} \ \{ \langle \text{id} \rangle \} \ \langle \text{CFAL} \rangle \} \\
| \ \{ \langle \text{CFAL} \rangle \ \langle \text{CFAL} \rangle \} \\
\]

\[
\{ \text{fun} \ \{x\} \ 0\} \ \{ + \ 1 \ \{ \text{fun} \ \{y\} \ 2\} \} \} \ \Rightarrow \ 0 \\
\{ \text{fun} \ \{x\} \ x\} \ \{ + \ 1 \ \{ \text{fun} \ \{y\} \ 2\} \} \} \ \Rightarrow \ \text{error}
\]
Implementing CFAL

Option #1: Run the FAE interpreter in PLAI Lazy!

; interp : CFAL DefrdSub -> CFAL-Value
(define (interp expr ds)
  ...
  [app (fun-expr arg-expr)
    (local [(define fun-val
      (interp fun-expr ds))
      (define arg-val
        (interp arg-expr ds))]
    (interp (closureV-body fun-val)
      (aSub (closureV-param fun-val)
        arg-val
        (closureV-ds fun-val)))))]

arg-val never used ⇒ interp call never evaluated
Implementing CFAL

Option #2: Use PLAI Racket and explicitly delay arg-expr interpretation

; interp : CFAL DefrdSub -> CFAL-Value
(define (interp expr ds)
  ...
  [app (fun-expr arg-expr)
     (local [[(define fun-val
                (interp fun-expr ds))
             (define arg-val
              (exprV arg-expr ds))]
         (interp (closureV-body fun-val)
              (aSub (closureV-param fun-val)
                  arg-val
                  (closureV-ds fun-val))))]]

where exprV is a new kind of CFAL-Value
(define-type CFAL-Value
  [numV (n number?)]
  [closureV (param symbol?)
    (body CFAL?)
    (ds DefrdSub?)])
  [exprV (expr CFAL?)
    (ds DefrdSub?)])
Forcing Evaluation for Number Operations

\[
\text{(interp } \{\{\text{fun } \{x\} \{+ 1 \ x\}\} 10\} \ (\text{mtSub}))
\]

⇒ error: expected \text{numV}, got exprV

\[
\text{(define } (\text{num-op } \text{op op-name } x \ y))
\]

\[
(\text{numV} (\text{op} (\text{numV-n} (\text{strict } x))
\quad (\text{numV-n} (\text{strict } y))))
\]

\[
\text{(define } (\text{num+ } x \ y) (\text{num-op } + \ '+ x \ y))
\]

\[
\text{(define } (\text{num- } x \ y) (\text{num-op } - \ '- x \ y))
\]

; \text{strict : CFAL-Value → CFAL-Value}
\[
\text{(define } (\text{strict } v))
\]

\[
(\text{type-case CFAL-Value } v
\quad [\text{exprV } (\text{expr ds}) (\text{strict } (\text{interp } \text{expr ds}))]
\quad [\text{else } v])
\]
Forcing Evaluation for Application

\[
\text{(interp} \quad \{\{\text{fun} \ \{f\} \ \{f \ 1\}\} \ \{\text{fun} \ \{x\} \ \{+ \ x \ 1\}\}\} \\
\text{(mtSub))}
\]

; interp : CFAL DefrdSub -> CFAL-Value
(define (interp expr ds)
...[app (fun-expr arg-expr)
    (local [(define fun-val
            (strict (interp fun-expr ds)))
            (define arg-val
            (exprV arg-expr ds))]
        (interp (closureV-body fun-val)
            (aSub (closureV-param fun-val)
                arg-val
                (closureV-ds fun-val))))])
Redundant Evaluation

\[
\{\text{fun } \{x\} \text{ and } \{\text{+ } x \text{ x} \text{ and } + x \text{ x}\}\} \quad \{- \text{ and } + 4 \text{ 5 }\text{ and } + 8 \text{ 9}\}\}
\]

How many times is \{+ 8 9\} evaluated?

Since the result is always the same, we’d like to evaluate
\{- + 4 5 \text{ and } + 8 9\} at most once
Caching Strict Results

```
(define-type CFAL-Value
    [numV (n number?)]
    [closureV (param symbol?)
        (body CFAL?)
        (ds DefrdSub?)]
    [exprV (expr CFAL?)
        (ds DefrdSub?)
        (value (box/c (or/c false CFAL-Value?)))]
)

; strict : CFAL-Value -> CFAL-Value
(define (strict v)
    (type-case CFAL-Value v
        [exprV (expr ds value-box)
            (if (not (unbox value-box))
                (local [(define v (strict (interp expr ds)))]
                    (begin
                        (set-box! value-box v)
                        v))
                (unbox value-box))]
        [else v]))
```
Fix Up Interpreter

```
(define (interp expr ds)
  ...
  [app ...
     (exprV arg-expr ds (box #f))
     ...])
```