CFAL = Lazy FAE

```
<CFAL>  ::=  <num>
          |  {+ <CFAL> <CFAL>}
          |  {- <CFAL> <CFAL>}
          |  <id>
          |  {fun {<id>} <CFAL>}
          |  {<CFAL> <CFAL>}
```

```
{{fun {x} 0} {+ 1 {fun {y} 2}}} ⇒ 0
{{fun {x} x} {+ 1 {fun {y} 2}}} ⇒ 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 Advanced 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`**

CFAL Values

```
(define-type CFAL-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body CFAL?)
            (ds DefrdSub?)]
  [exprV (expr CFAL?)
         (ds DefrdSub?)])
```

Forcing Evaluation for Number Operations

```
(interp {{fun {x} {+ 1 x}} 10} (mtSub))
```

⇒ *error: expected numV, got exprV*

```
(define (num-op op op-name x y)
  (numV (op (numV-n (strict x))
            (numV-n (strict y)))))

(define (num+ x y) (num-op + '+ x y))
(define (num- x y) (num-op - '- x y))

; strict : CFAL-Value -> CFAL-Value
(define (strict v)
  (type-case CFAL-Value v
    [exprV (expr ds) (strict (interp expr ds))]
    [else v]))
```

Forcing Evaluation for Application

```
(interp {{fun {f} {f 1}} {fun {x} {+ x 1}}}
        (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

```
{{fun {x} {+ {+ x x} {+ x x}}}
 {- {+ 4 5} {+ 8 9}}}
```

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

Since the result is always the same, we'd like to evaluate `{- {+ 4 5} {+ 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-of (false-or CFAL-Value?)))])

; strict : CFAL-Value -> CFAL-Value
(define (strict v)
  (type-case CFAL-Value v
    [exprV (expr ds value-box)
           (if (false? (unbox value-box))
               (local [(define v (strict (interp expr ds)))]
                 (begin
                   (set-box! value-box v)
                   v))
               (unbox value-box))]
    [else v]))
```

Etc.

```
(define (false? v)
  (and (boolean? v)
       (not v)))

(define (box-of p)
  (lambda (v)
    (and (box? v)
         (p (unbox v)))))

(define (false-or p)
  (lambda (v)
    (or (false? v)
        (p v))))

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