Part I
Racket vs. Algebra

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

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

\[ g(z) = ... \]

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

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
Part 2
Lazy Evaluation in DrRacket

The **plai-lazy** package adds support for

```
#lang plai-lazy
```

For coverage reports:

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

(Works for both eager and lazy languages)

- Black means evaluated at least once
- **Orange** means not yet evaluated
- Normal coloring is the same as all black
Part 3
Doesn’t work because result of \texttt{set-box!} is never used:

\begin{verbatim}
(define (interp a env)
  (type-case ExprC a
    ...
    [letrecC (n rhs body)
      (let ([b (box (numV 42)))]
        (let ([new-env (extend-env
                         (bind n b)
                         env)])
          (begin
            (set-box! b (interp rhs new-env))
            (interp body new-env))))])
\end{verbatim}
letrec Interpreter in plai-lazy

Working implementation is more direct:

```
(define (interp a env)
  (type-case ExprC a
    ...
    [letrecC (n rhs body)
        (letrec ([new-env
                    (extend-env
                      (bind n (interp rhs new-env))
                      env)])
          (interp body new-env))]]))
```
Part 4
Lazy Language

\[
<Expr> ::= <Num> \\
| <Sym> \\
| \{ + <Expr> <Expr> \} \\
| \{ * <Expr> <Expr> \} \\
| \{ lambda \{<Sym>\} <Expr> \} \\
| \{ <Expr> <Expr> \}
\]

\{ lambda \{x\} 0 \{ + 1 \{ lambda \{y\} 2 \} \} \} \Rightarrow 0
\{ lambda \{x\} x \{ + 1 \{ lambda \{y\} 2 \} \} \} \Rightarrow error

\{ let \{ [x \{ + 1 \{ lambda \{y\} 2 \} ] \} 0 \} \Rightarrow 0 \}
Part 5
Implementing Laziness

**Option #1:** Run the interpreter in `plai-lazy`!

```
(define (interp a env)
  (type-case ExprC a
    ...
    [appC (fun arg)
      (type-case Value (interp fun env)
        [closV (n body c-env)
          (interp body
            (extend-env
              (bind n (interp arg env))
              c-env))]
        [else (error 'interp "not a function")]]))

n never used ⇒ interp call never evaluated
```
Implementing Laziness

Option #2: Use \texttt{plai-typed} and explicitly delay \texttt{arg}
interpretation

\begin{verbatim}
(define (interp a env)
  (type-case ExprC a
    ...
    [appC (fun arg)
      (type-case Value (interp fun env)
        [closV (n body c-env)
          (interp body
            (extend-env
              (bind n (suspendV arg env))
            c-env))]
        [else (error 'interp "not a function")])))

where \texttt{suspendV} is a new kind of \texttt{Value}
\end{verbatim}
(define-type Value
  [numV (n : number)]
  [closV (arg : symbol)
    (body : ExprC)
    (env : Env)]
  [suspendV (body : ExprC)
    (env : Env)])
Part 6-7
Forcing Evaluation for Number Operations

\[
\text{interp} \quad \{ \{ \text{lambda} \; \{ x \} \; \{ + \; 1 \; x \} \} \; 10 \} \quad \text{mt-env}
\]

\[ \Rightarrow \text{error: expected numV, got suspendV} \]

\[ \Rightarrow \quad \text{interp} \quad \{ + \; 1 \; x \}
\]

\[
\text{extend-env} \quad \text{bind} \; \{ x \}
\]

\[
\text{suspendV} \quad 10 \quad \text{mt-env}
\]

\[ \text{mt-env} \]

29-31
Forcing Evaluation for Number Operations

\[(\text{interp} \{\{\lambda x. (+ 1 x)\} 10\} \text{ mt-env})\]

\[\Rightarrow \text{error: expected numV, got suspendV}\]

\[(\text{define} \ (\text{interp} \ [a : \text{ExprC}] \ [\text{env} : \text{Env}]) : \text{Value} \ (%\text{type-case ExprC a}
\ldots
\ [\text{plusC} \ (l \ r) \ (\text{num+} \ (\text{strict} \ (\text{interp} \ l \ \text{env})))\ (\text{strict} \ (\text{interp} \ r \ \text{env})))])\]

\[(\text{define} \ (\text{strict} \ [v : \text{Value}]) : \text{Value} \ (%\text{type-case Value v}
\ [\text{suspendV} \ (b \ e) \ (\text{strict} \ (\text{interp} \ b \ e))]\ [\text{else} \ v]))\]
Part 8
Forcing Evaluation for Application

\[
\text{(interp } \{ \lambda \{ f \} \{ f \ 1 \} \} \{ \lambda \{ x \} \{ + \ x \ 1 \} \}) \text{)
}\]

\[
\Rightarrow
\]

\[
\text{(interp } \{ f \ 1 \})
\]

\[
\text{(extend-env}
\text{(bind 'f}
\text{(suspendV}
\text{(lambda } \{ x \} \{ + \ x \ 1 \})
\text{mt-env))}
\text{mt-env))}
\]
Forcing Evaluation for Application

```
(interp \{\{\lambda f \{f 1\}\}\ \{\lambda x \{+ x 1\}\}\})
```

```
(define (interp a env)
  (type-case ExprC a

  ...

  [appC (fun arg)
   (type-case Value (strict (interp fun env)))
    [closV (n body c-env)
     (interp body
      (extend-env
       (bind n (suspendV arg env))
       c-env))]
    [else (error 'interp "not a function")]]))
```

Part 9
Redundant Evaluation

\[
\{\{\text{lambda } \{x\} \{+ \{+ x x\} \{+ x x\}\}\}\}
\{- \{+ 4 5\} \{+ 8 9\}\}\}
\]

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 Value
... [suspendV (body : ExprC)
    (env : Env)
    (done : (boxof (optionof Value)))]
}
(define (strict [v : Value]) : Value
  (type-case Value v
    [suspendV (b e d)
      (type-case (optionof Value) (unbox d)
        [none ()
          (let ([v (strict (interp b e))])
            (begin
              (set-box! d (some v) v))]
        [some (v) v]])
    [else v]))
(define (interp a env)
  ....
  [appC (fun arg)
   ... (suspendV arg env (box (none))) ...]"
Part 10
Terminology

*Call-by-value* means eager

Racket, Java, C, Python...

*Call-by-name* means lazy, no caching of results

... which is impractical

*Call-by-need* means lazy, with caching of results

Haskell, Clean...
Terminology

*Normal order vs Applicative order*

...good terms to avoid