Part 1

Racket vs. Algebra

```
(+ (* 4 3) (- 8 7)) ⇒ (+ 12 (- 8 7)) ⇒ (+ 12 1)
```

Racket vs. Algebra

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

```
(+ (* 4 3) (- 8 7)) ⇒ (+ 12 (- 8 7)) ⇒ (+ 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

$$f(x, y) = x$$

$$g(z) = \ldots$$

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

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

`letrec` Interepreter in `plai-lazy`

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

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

`letrec` Interepreter 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}}}  ⇒  0
{{lambda {x} x} {+ 1 {lambda {y} 2}}}  ⇒  error
```

```
{let {[x {+ 1 {lambda {y} 2}}]}
  0}
⇒ 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 **`plai-typed`** and explicitly delay **`arg`** interpretation

```
(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 (delay arg env))
                            c-env))]
            [else (error 'interp "not a function")])]))
```

Thunks and Bindings

```
(define-type Thunk
  [delay (body : ExprC)
         (env : Env)])

(define-type Binding
  [bind (name : symbol)
        (val : Thunk)])
```

Implementing Laziness

```
(define (interp a env)
  (type-case ExprC a
    ...
    ...
    ...
    [appC (fun arg)
          ...
          (extend-env
           (bind n (delay arg env))
           c-env)
          ...]))
```

Implementing Laziness

```
(define (interp a env)
  (type-case ExprC a
    ...
    [idC (s) (force (lookup s env))]
    ...
    [appC (fun arg)
          ...
          (extend-env
           (bind n (delay arg env))
           c-env)
          ...]))
```

Implementing Laziness

```
(define (interp a env)
  (type-case ExprC a
    ...
    [idC (s) (force (lookup s env))]
    ...
    [appC (fun arg)
          ...
          (extend-env
           (bind n (delay arg env))
           c-env)
          ...]))

(define (force [t : Thunk]) : Value
  (type-case Thunk t
    [delay (b e) (interp b e)]))
```

Part 6

Redundant Evaluation

```
{{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 Force Results

```
(define-type Thunk
  [delay (body : ExprC)
         (env : Env)
         (done : (boxof (optionof Value)))])
```

Fix Up Interpreter

```
(define (interp a env)
  ....
  [appC (fun arg)
        ... (delay arg env (box (none))) ...])
```

Caching Force Results

```
(define (force [t : Thunk]) : Value
  (type-case Thunk t
    [delay (b e) (interp b e)]))
```

⇒

```
(define (force [t : Thunk]) : Value
  (type-case Thunk t
    [delay (b e d)
      (type-case (optionof Value) (unbox d)
        [none ()
              (let ([v (interp b e)])
                (begin
                  (set-box! d (some v))
                 v))]
        [some (v) v])]))
```

Part 7

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