Part I
Shplait vs. Algebra

\[4 \times 3 + 8 - 7 \Rightarrow 12 + 8 - 7 \Rightarrow 12 + 1\]
Shplait vs. Algebra

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

\[ 4 \times 3 + 8 - 7 \Rightarrow 12 + 8 - 7 \Rightarrow 12 + 1 \]
Shplait vs. Algebra

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

\[ 4 \times 3 + 8 - 7 \Rightarrow 12 + 8 - 7 \Rightarrow 12 + 1 \]

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

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

\[ g(z) = \ldots \]

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

then we can go straight to

\[ 17 \]

because the result of all the \( g \) calls will not be used
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

```python
fun layout_text(txt, w, h):
    def lines:
        // lots of work to flow a paragraph
        ....
        make_pict(w,
                   h,
                   fun (dc, x, y):
                       // draw paragraph lines
                       ....)

    ....
    def speech = layout_text("Four score...", 800, 600)

    ....
pict_width(speech)
```
Avoiding Unnecessary Work

fun read_all_chars(f):
    if is_at_eof(f)
        | []
        | cons(read_char(f), read_all_chars(f))

def content = read_all_chars(open_file(user_file))

if first(content) == "#"
    | process_file(rest(content))
    | error(('#'parser, "not a valid file"))
Recursive Definitions

fun numbers_from(n):
    cons(n, numbers_from(add1(n)))

def nonneg = numbers_from(0)
list_get(nonneg, 10675)
Lazy Evaluation

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

• An expression is evaluated when it is encountered
Lazy Evaluation

Languages like Shplait, 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 Shplait

Use

```
#lang shplait
~lazy
```

to run a Shplait program with lazy evaluation
Lazy Evaluation in Shplait

For coverage reports in DrRacket:

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** Interpreter in Lazy Shplait

Doesn’t work because result of `set_box` is never used:

```plaintext
fun interp(a, env):
  match a
  | ...
  | letrecE(n, rhs, body):
    let b = box(none()):
    let new_env = extend_env(bind(n, b),
                            env):
      set_box(b, some(interp(rhs, new_env)))
    interp(body, new_env)
```

letrec Interpreter in Lazy Shplait

Working implementation is more direct:

```haskell
fun interp(a, env):
    match a
    | ...
    | letrecE(n, rhs, body):
        letrec new_env = extend_env(bind(n, interp(rhs, new_env)),
                                    env):
            interp(body, new_env)
```

Part 4
Lazy Language

\[
\textit{<Exp>} ::= \textit{<Int>} \\
| \textit{<Symbol>} \\
| \textit{<Exp>} + \textit{<Exp>} \\
| \textit{<Exp>} * \textit{<Exp>} \\
| \text{fun}\ (\textit{<Symbol>})\ :\ \textit{<Exp>} \\
| \textit{<Exp>}\ (\textit{<Exp>})
\]
Lazy Language

\[
<\text{Exp}> ::= <\text{Int}> \\
| <\text{Symbol}> \\
| <\text{Exp}> + <\text{Exp}> \\
| <\text{Exp}> * <\text{Exp}> \\
| \text{fun} (<\text{Symbol}>): <\text{Exp}> \\
| <\text{Exp}> (<\text{Exp}>)
\]

\[
\begin{align*}
\text{(fun } x : 0)(1 + (\text{fun } y : 2)) & \Rightarrow 0 \\
\text{(fun } x : x)(1 + (\text{fun } y : 2)) & \Rightarrow \text{error}
\end{align*}
\]
Lazy Language

\[
\begin{align*}
<\text{Exp}> & ::= <\text{Int}> \\
& \mid <\text{Symbol}> \\
& \mid <\text{Exp}> + <\text{Exp}> \\
& \mid <\text{Exp}> \times <\text{Exp}> \\
& \mid \text{fun} (<\text{Symbol}>): <\text{Exp}> \\
& \mid <\text{Exp}>(<\text{Exp}>)
\end{align*}
\]

\[
\begin{align*}
\text{(fun (x): 0)(1 + (fun (y): 2))} & \Rightarrow 0 \\
\text{(fun (x): x)(1 + (fun (y): 2))} & \Rightarrow \text{error}
\end{align*}
\]

\[
\begin{align*}
\text{let x = 1 + (fun (y): 2):} \\
& 0 \\
& \Rightarrow 0
\end{align*}
\]
Part 5
Implementing Laziness

Option #1: Run the interpreter in `shplait ~lazy!`
Implementing Laziness

Option #1: Run the interpreter in `shplait ~lazy`!

```python
fun interp(a, env):
    match a
    | ... 
    | appE(fn, arg):
        match interp(fn, 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 Shplait and explicitly delay \texttt{arg} interpretation
Implementing Laziness

**Option #2:** Use Shplait and explicitly delay `arg` interpretation

```python
fun interp(a, env):
    match a
    | ... 
    | appE(fn, arg):
        match interp(fn, 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

type Thunk
| delay(arg :: Exp,
    env :: Env)

type Binding
| bind(name :: Symbol,
    val :: Thunk)
Implementing Laziness

```plaintext
fun interp(a, env):
    match a
    | ...  
    ... 
    | ... 
    | ... 
    | appE(fn, arg):
    ... 
      extend_env(bind(n, delay(arg, env)),
                 c_env)
    ... 
```
Implementing Laziness

```python
fun interp(a, env):
    match a
    | ... |
    | idE(s): force(lookup(s, env)) |
    | ... |
    | appE(fn, arg):
    ... 
    extend_env(bind(n, delay(arg, env)),
               c_env)
    ... 
    ... 
```
Implementing Laziness

fun interp(a, env):
    match a
    | ...  
    | idE(s): force(lookup(s, env))
    | ...  
    | appE(fn, arg):
        ...
        extend_env(bind(n, delay(arg, env)), c_env)
        ...

fun force(t :: Thunk) :: Value:
    match t
    | delay(arg, env): interp(arg, env)
Part 6
Redundant Evaluation
Redundant Evaluation

\[
\text{(fun } x \text{): } x + x + x + x) (4 + 5 - 8 + 9)
\]

How many times is \(8 + 9\) evaluated?
Redundant Evaluation

\[(\textsf{fun } (x) : x + x + x + x) (4 + 5 - 8 + 9)\]

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 Force Results

type Thunk
| delay(arg :: Exp, 
  env :: Env, 
  done :: Boxof(Optionof(Value)))
fun interp(a, env):
    ....
    | appE(fn, arg):
    .... delay(arg, env, box(none()))  ....
Caching Force Results

fun force(t :: Thunk) :: Value:
  match t
  | delay(arg, env): interp(arg, env)
Caching Force Results

fun force(t :: Thunk) :: Value:
  match t
  | delay(arg, env): interp(arg, env)

⇒

fun force(t :: Thunk) :: Value:
  match t
  | delay(arg, env, done):
    match unbox(done)
    | none():
      let v = interp(arg, env):
      set_box(done, some(v))
      v
    | some(v): v
Part 7
Terminology
Terminology

*Call-by-value* means eager

Shplait, Java, C, Python...
**Terminology**

*Call-by-value* means eager

Shplait, Java, C, Python...

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

... which is impractical
Terminology

**Call-by-value** means eager

Shplait, 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