Reminder

Mid-Term 1 is Tuesday next week

(No homework will be assigned this week)
letrec fact = proc(n) if n then *(n, (fact - (n, 1))) else 1 in (fact 10)
Recap: Recursive Environments

double box means a recursively extended environment

letrec fact = proc(n) if n then *(n, (fact - (n, 1))) else 1
in (fact 10)
Recap: Recursive Environments

\[
\text{letrec } \text{fact } = \text{proc}(n) \text{ if } n \text{ then } *(n, (\text{fact }-(n, 1))) \text{ else } 1 \\
\text{in } (\text{fact } 10)
\]
Recap: Recursive Environments

every lookup of fact generates a closure

letrec fact = proc(n) if n then *(n, (fact -(n, 1)))) else 1
in (fact 10)
Recap: Recursive Environments

\[
\text{fact } n \text{ if } n \text{ then } *(n, (\text{fact} -(n, 1))) \text{ else } 1
\]

\[
\text{n if } n \text{ then } *(n, (\text{fact} -(n, 1))) \text{ else } 1
\]

\[
\text{letrec fact } = \text{ proc}(n) \text{ if } n \text{ then } *(n, (\text{fact} -(n, 1))) \text{ else } 1 \text{ in } (\text{fact 10})
\]
Recap: Recursive Environments

letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1 in (fact 10)
Recap: Recursive Environments

\[
\text{letrec } \text{fact} = \text{proc(n) if n then } *(n, (\text{fact} -(n, 1))) \text{ else 1 in (fact 10)}
\]
Recap: Recursive Environments

letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1 in (fact 10)
Recap: Recursive Environments

\[
\text{letrec fact } = \text{ proc(n) if n then } \ast(n, (\text{fact } - (n, 1))) \text{ else 1 in (fact 10)}
\]
Recap: Recursive Environments

\[
\text{letrec } \text{fact} = \text{proc}(n) \text{ if } n \text{ then } *(n, (\text{fact } -(n, 1))) \text{ else } 1 \\
\text{ in } (\text{fact } 10)
\]
Recap: Recursive Environments

letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
in (fact 10)
Recap: Recursive Environments

letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1 in (fact 10)
Another Approach to Recursive Closures

letrec fact = proc(n) if n then *(n, (fact -(n, 1)))) else 1
in (fact 10)
Another Approach to Recursive Closures

create an environment with a dummy value...

\[
\text{letrec fact } = \text{proc}(n) \text{ if } n \text{ then } *(n, (\text{fact } - (n, 1))) \text{ else } 1 \\
\text{in (fact 10)}
\]
Another Approach to Recursive Closures

letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
    in (fact 10)

create the closure using the environment...
Another Approach to Recursive Closures

letrec fact = proc(n) if n then *(n, (fact -(n, 1)))) else 1 in (fact 10)
Another Approach to Recursive Closures

letrec fact = proc(n) if n then *(n, (fact - (n, 1))) else 1 in (fact 10)
Another Approach to Recursive Closures

letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1 in (fact 10)
Another Approach to Recursive Closures

letrec fact = proc(n) if n then *(n, (fact -(n, 1)))) else 1 in (fact 10)

an advantage: closure is only created once
Modifying Environments

• Nothing in Scheme so far supports *modifying* a value

• So we need evaluation rules to support *vectors* and *vector update*
Evaluation of Vector Expressions

- Unlike `cons`, `vector` does not create a value.
- Instead, it's treated like local functions used to be:

```
... (let ([v (vector 1 2 3)]) (vector-ref v 0))
→
... (define vec\_1743 (vector 1 2 3))
(let ([v vec\_1743]) (vector-ref v 0))
→
... (define vec\_1743 (vector 1 2 3))
(vector-ref vec\_1743 0)
→
... (define vec\_1743 (vector 1 2 3))
1
```
Evaluation of Vector Expressions

- The reason for this definition of vector is to enable vector-set! ...

(let ([v (vector 1 2 3)])
  (begin
    (vector-set! v 0 5)
    (vector-ref v 0)))
→

... (define vec_{1743} (vector 1 2 3))
(let ([v vec_{1743}])
  (begin
    (vector-set! v 0 5)
    (vector-ref v 0)))
→

... (define vec_{1743} (vector 1 2 3))
(begin
  (vector-set! vec_{1743} 0 5)
  (vector-ref vec_{1743} 0))
→

... (define vec_{1743} (vector 5 2 3))
(vector-ref vec_{1743} 0)
→

... (define vec_{1743} (vector 5 2 3))
5
Begin Expressions

• **begin** evaluates a sequence of expressions, in order

• **lambda** and **let** always supply an implicit **begin**

\[
\text{let } (...) \ <expr>_1 \ ... \ <expr>_n) = (\text{let } (...) \ (\text{begin } <expr>_1 \ <expr>_n))
\]

\[
\text{lambda } (...) \ <expr>_1 \ ... \ <expr>_n) = (\text{lambda } (...) \ (\text{begin } <expr>_1 \ <expr>_n))
\]
Changing Recursive Environment Extension

Now we can change `extend-env-recursively` to use `vector-set!`

Go back to just two datatype variants

```scheme
(define-datatype environment environment? 
    (empty-env-record)
    (extended-env-record
        (syms (list-of symbol?))
        (vals vector?)
        (env environment?))

(implement in DrScheme)
```
What if we change \( z \) to \( x \)?

\[
\begin{align*}
\text{let } x &= 1 \quad \text{y} = 2 \\
\text{in } \text{let } f &= \text{proc } (z) \ + (z, \ y) \\
\text{in } (f \ y)
\end{align*}
\]
Shape of the environment and location of the argument is unchanged

- argument is always first in first frame
- \( y \) is always second in second frame

```
let x = 1  y = 2
in let f = proc (x) + (x, y)
    in (f y)
```
Still true if \( f \) is called from a more complex environment

\[
\text{let } x = 1 \quad y = 2 \\
\text{in let } f = \text{proc } (x) + (x, y) \\
\text{in } +((f \ y), \text{let } w = 5 \text{ in } (f \ 3))
\]
Compilation

So why waste time searching the environment on every variable access?

A compiler can determine the *lexical offset* for each variable statically.

Terminology:

- A *compiler* translates a program from language $X$ to language $Y$.
- An *interpreter* executes a program in language $X$. 


Compilation of Variable Accesses

- We'll write a compiler that transforms

```plaintext
let x = 1  y = 2
  in let f = proc (x) +(x, y)
       in (f x)
```


to

```plaintext
let _ = 1  _ = 2
  in let _ = proc (_,) +(<0,0>, <1,1>)
      in (<0,0> <1,0>)
```

- We'll also need an interpreter for the new language