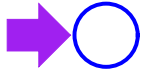


## **Reminder**

Mid-Term 1 is Tuesday next week

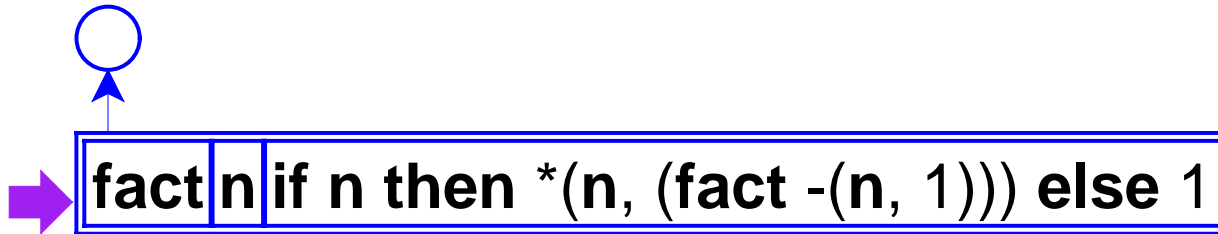
(No homework will be assigned this week)

# Recap: Recursive Environments



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

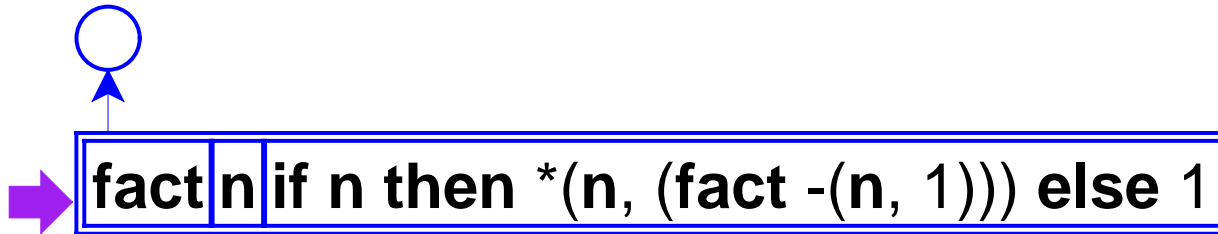
## Recap: Recursive Environments



*double box means a recursively extended environment*

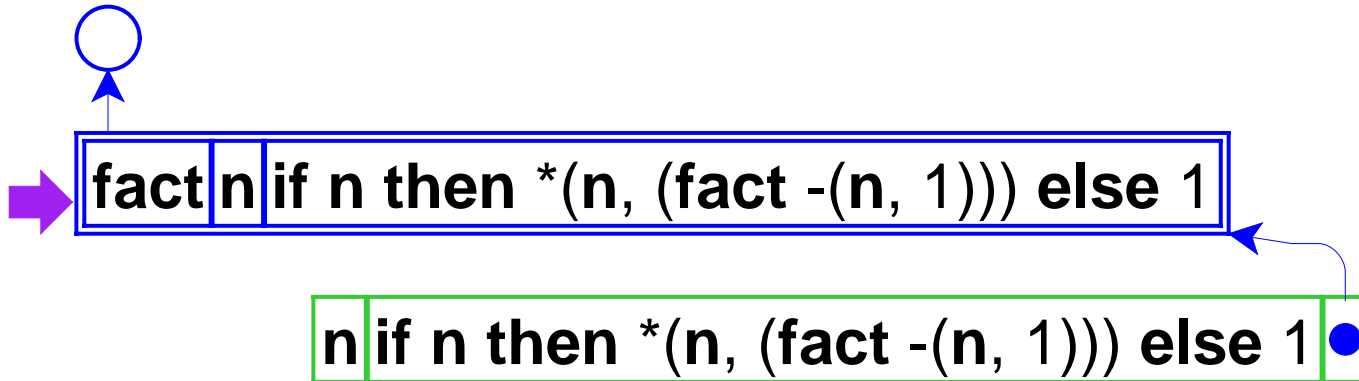
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
in (fact 10)
```

# Recap: Recursive Environments



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

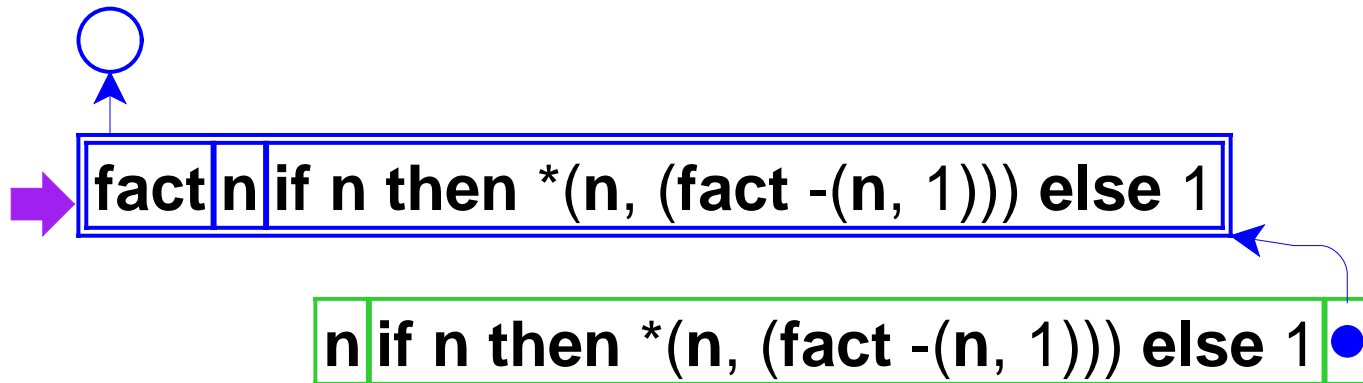
## Recap: Recursive Environments



*every lookup of **fact**  
generates a closure*

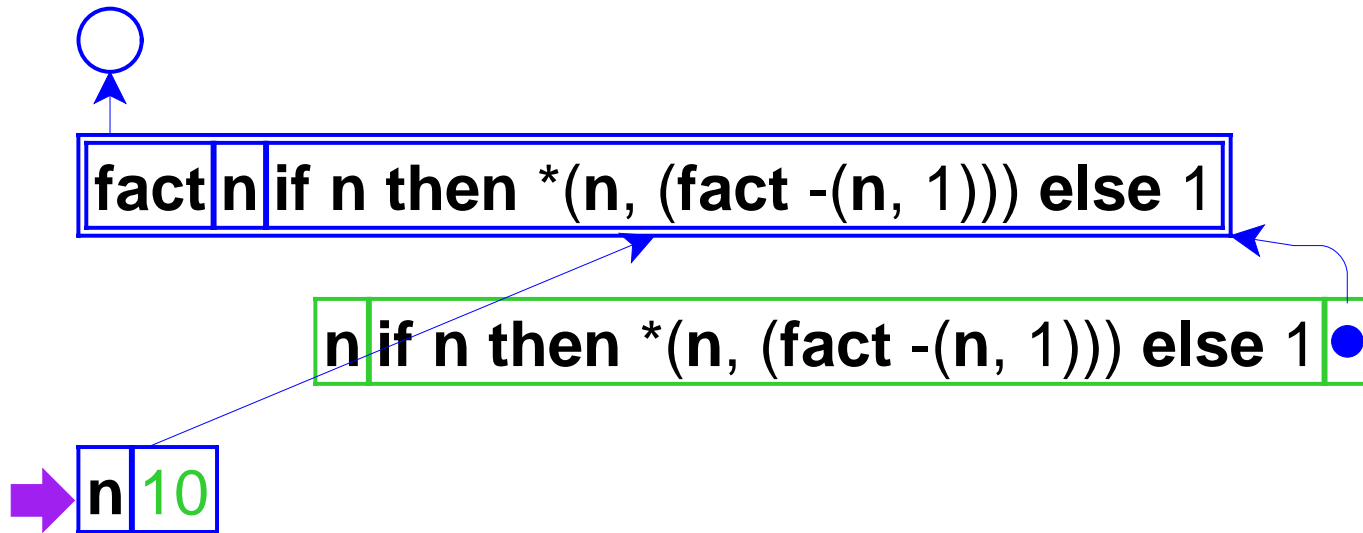
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
in (fact 10)
```

# Recap: Recursive Environments



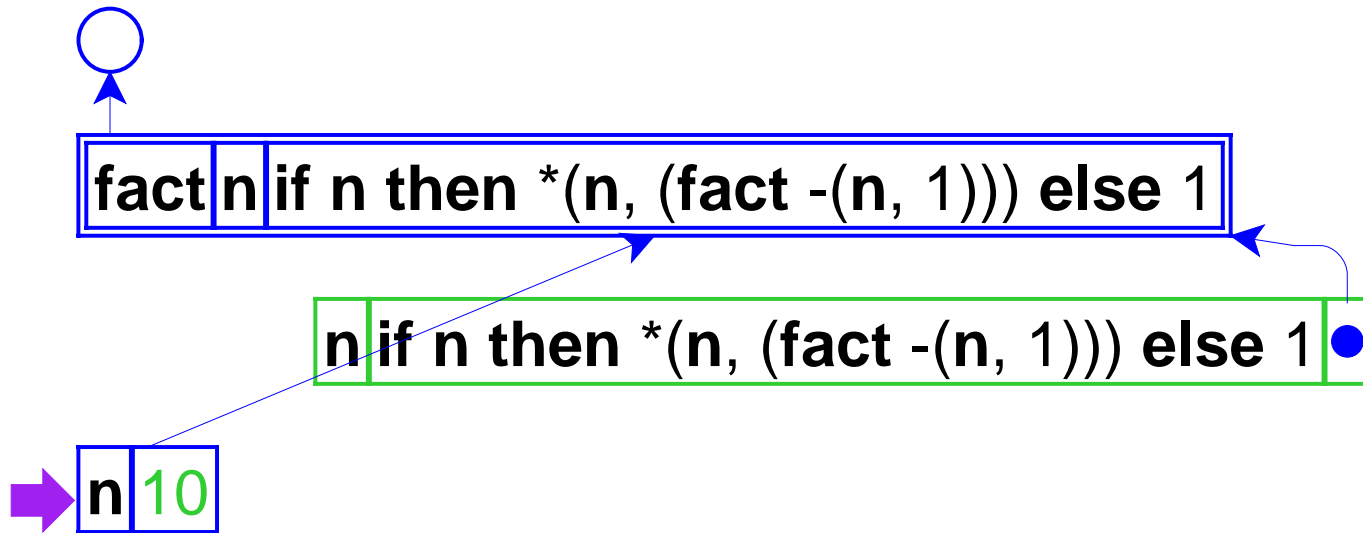
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
in (fact 10)
```

# Recap: Recursive Environments



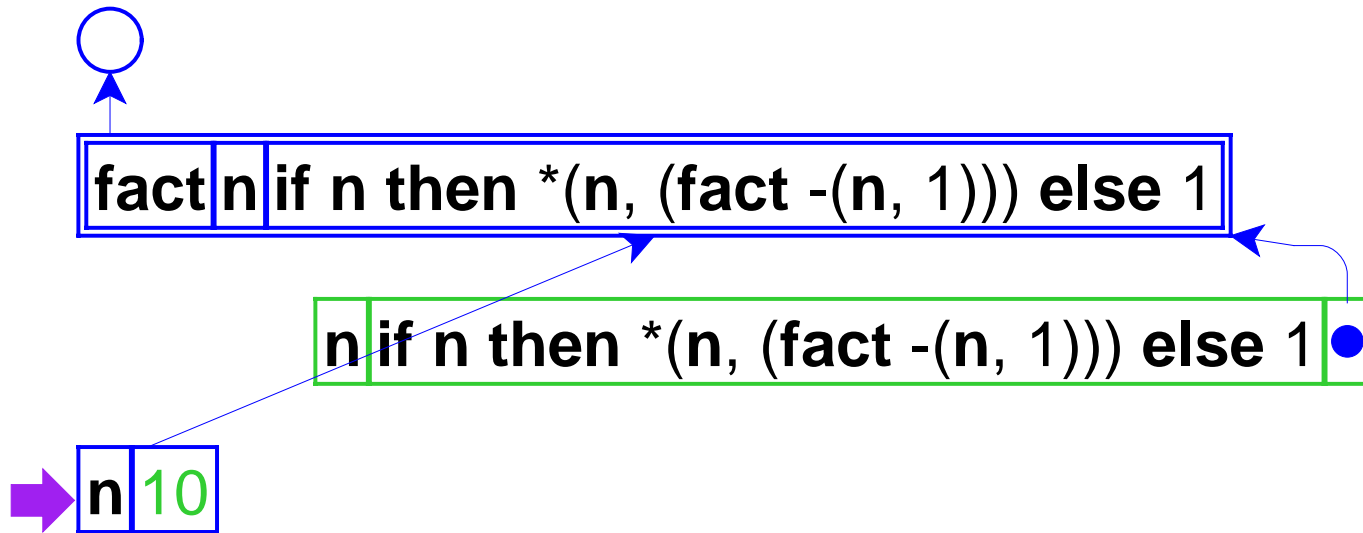
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
in (fact 10)
```

# Recap: Recursive Environments



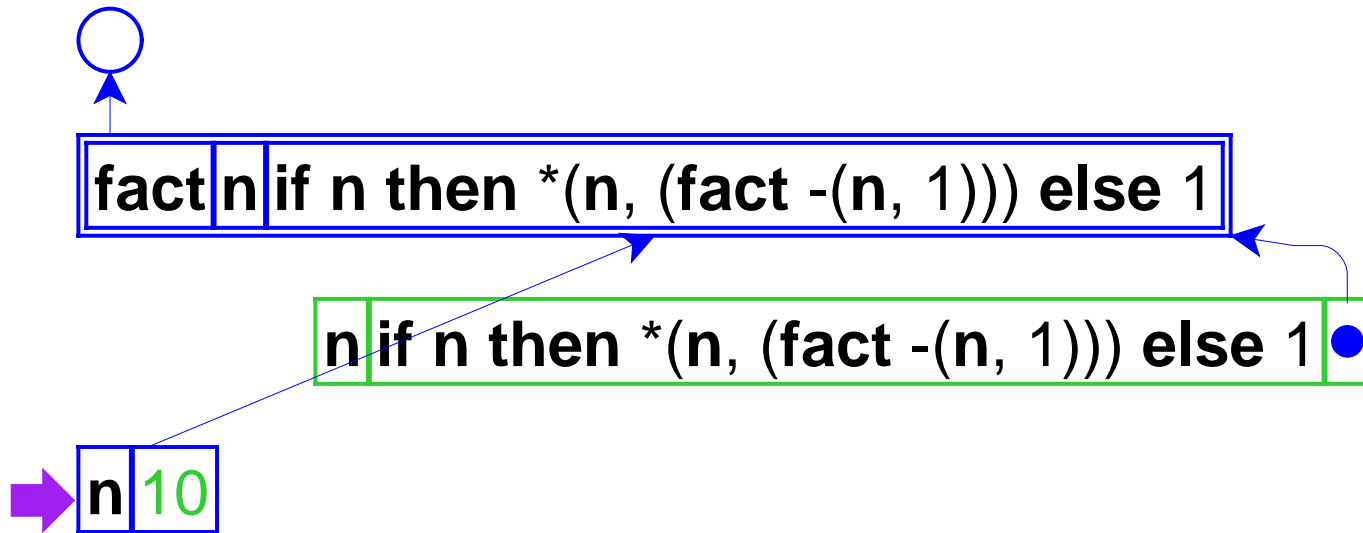
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
in (fact 10)
```

# Recap: Recursive Environments



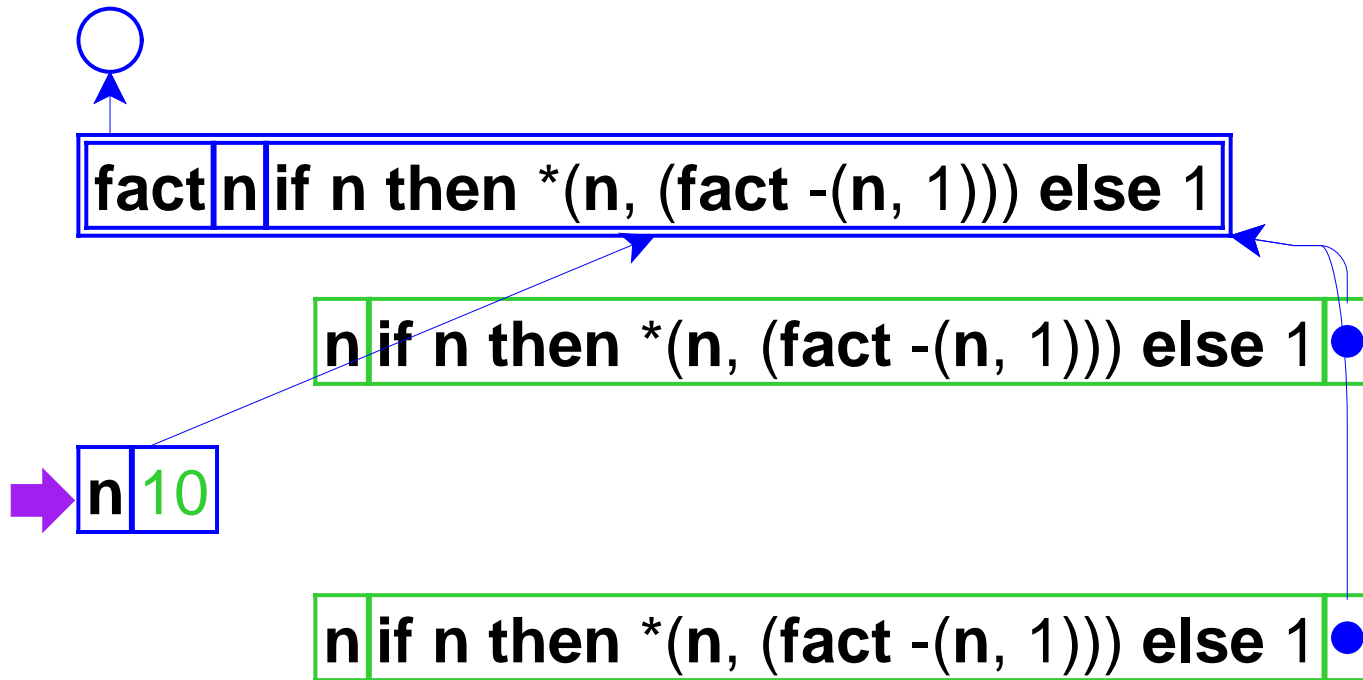
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
in (fact 10)
```

# Recap: Recursive Environments



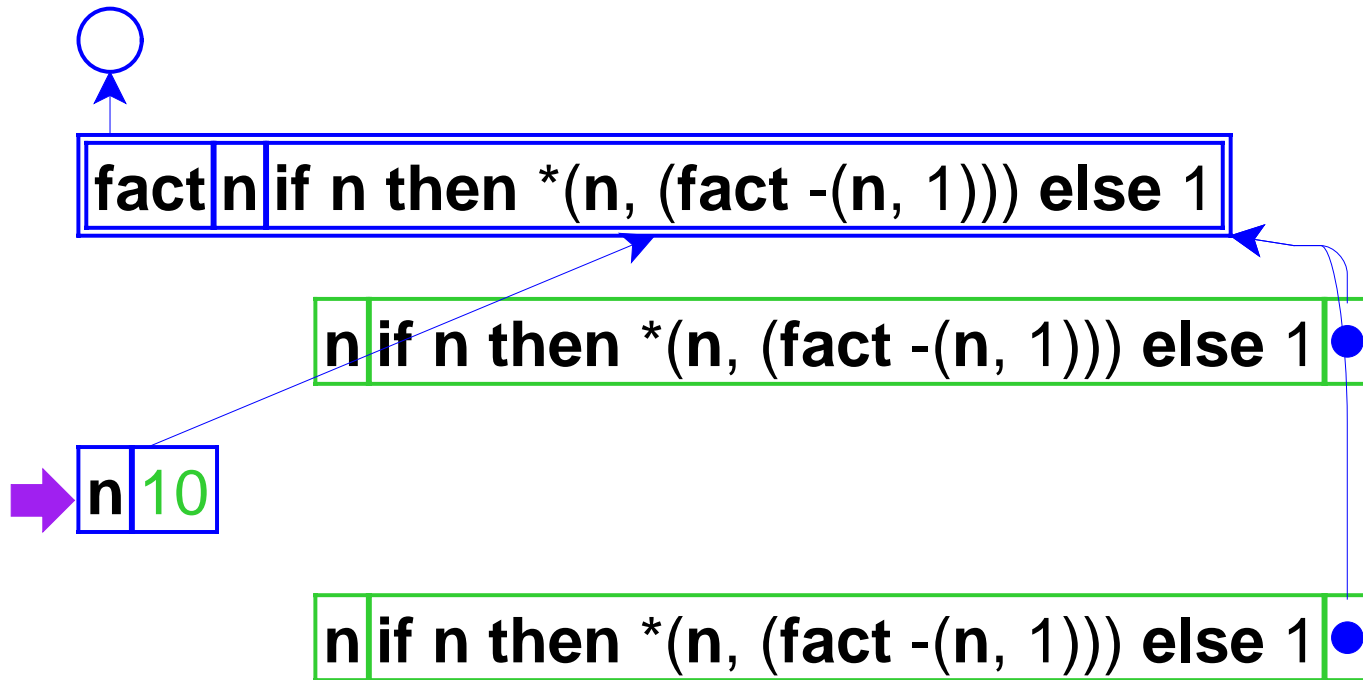
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
in (fact 10)
```

# Recap: Recursive Environments



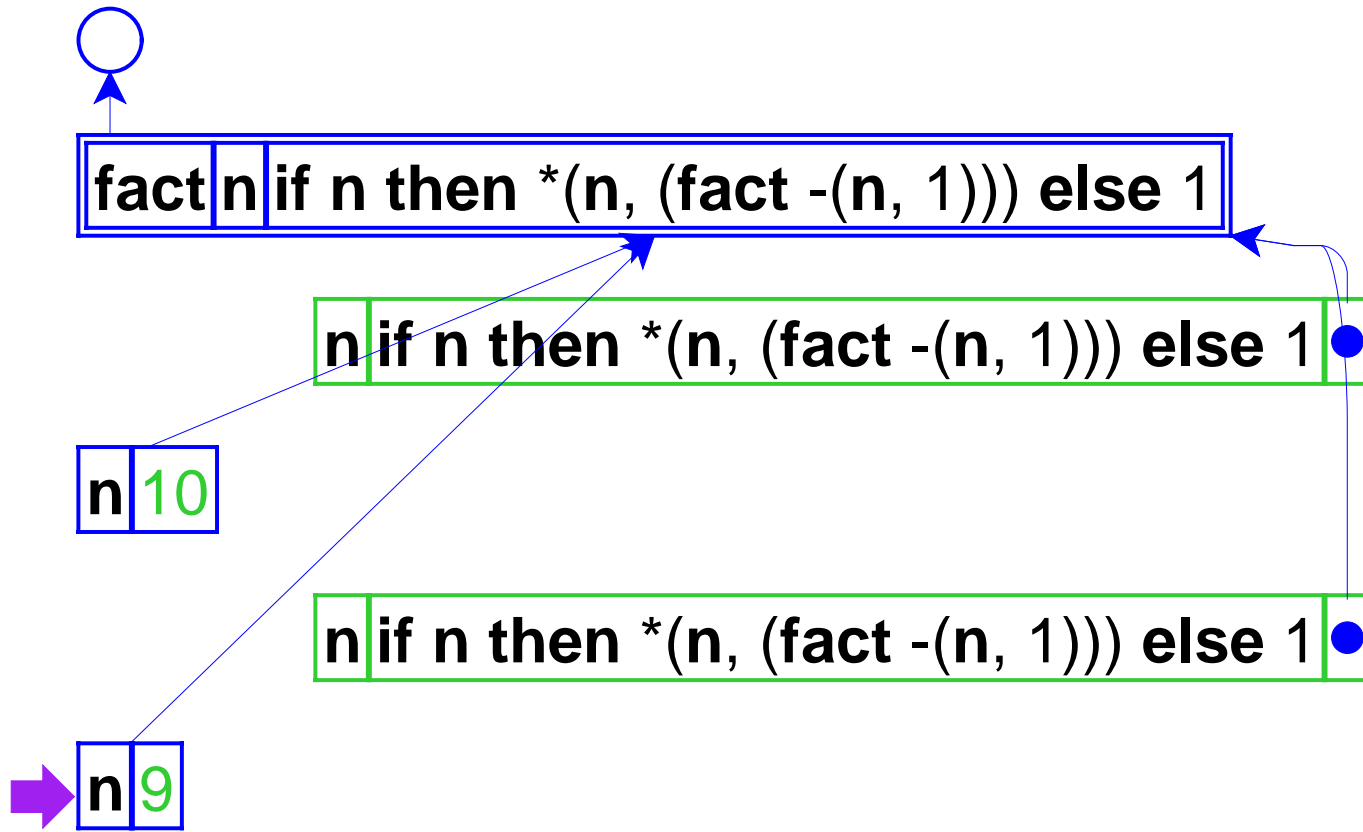
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
in (fact 10)
```

# Recap: Recursive Environments



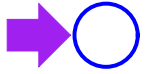
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
in (fact 10)
```

# Recap: Recursive Environments



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

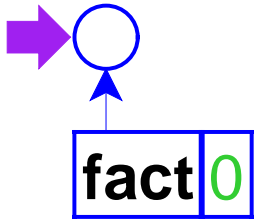
# Another Approach to Recursive Closures



*alternate approach...*

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

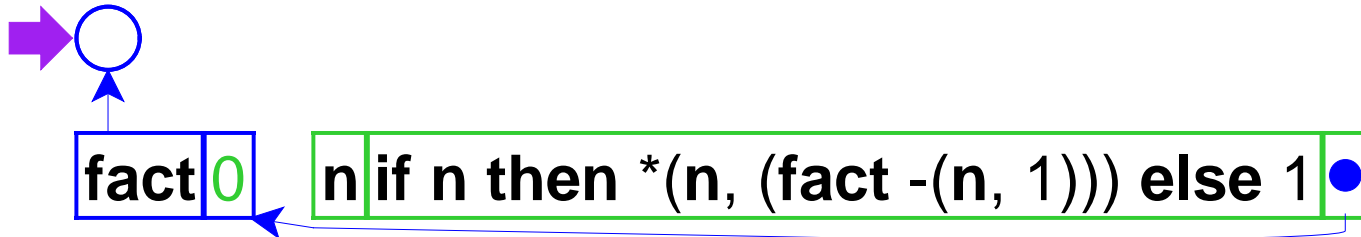
## Another Approach to Recursive Closures



*create an environment  
with a dummy value...*

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

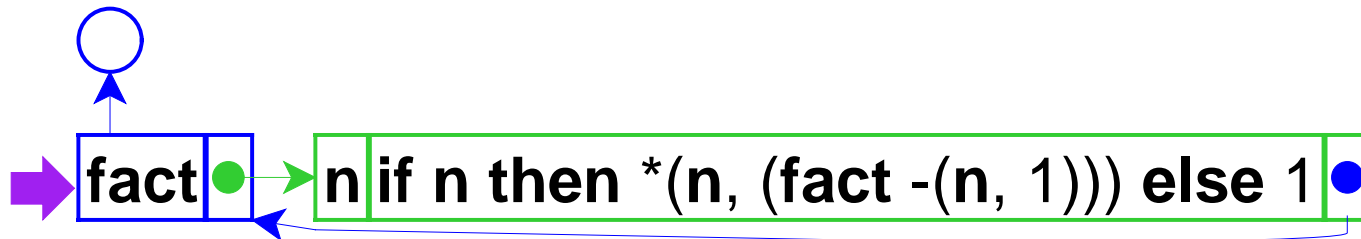
## Another Approach to Recursive Closures



*create the closure using  
the environment...*

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

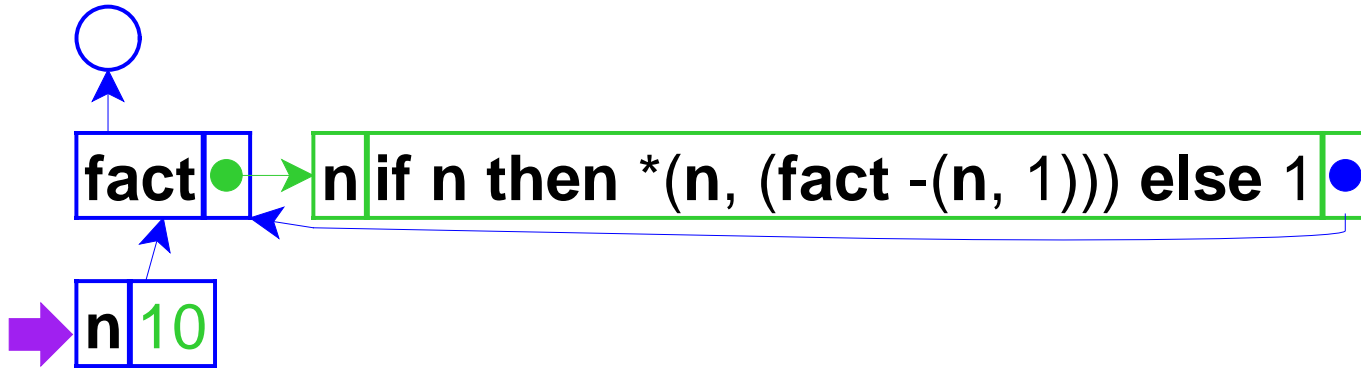
## Another Approach to Recursive Closures



*then  
modify  
the environment  
to fix it up*

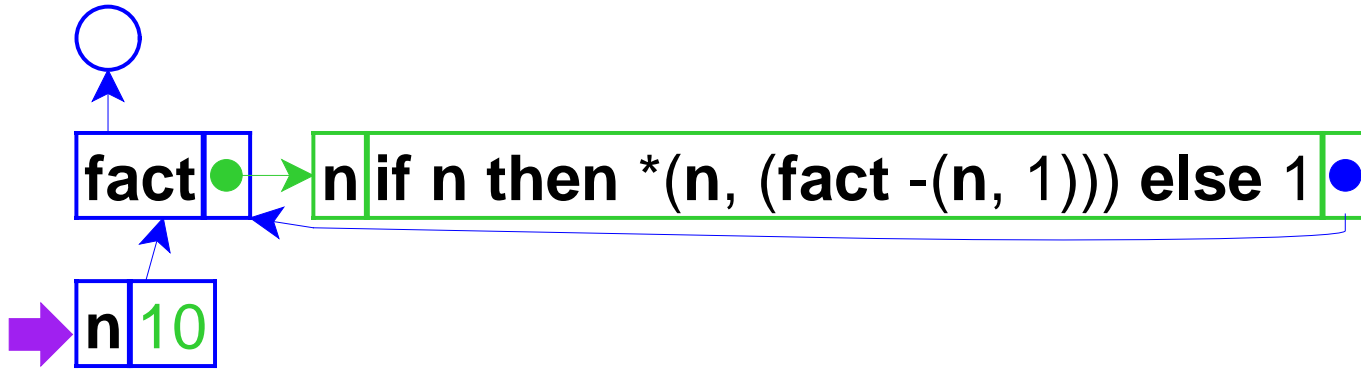
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1  
in (fact 10)
```

## Another Approach to Recursive Closures



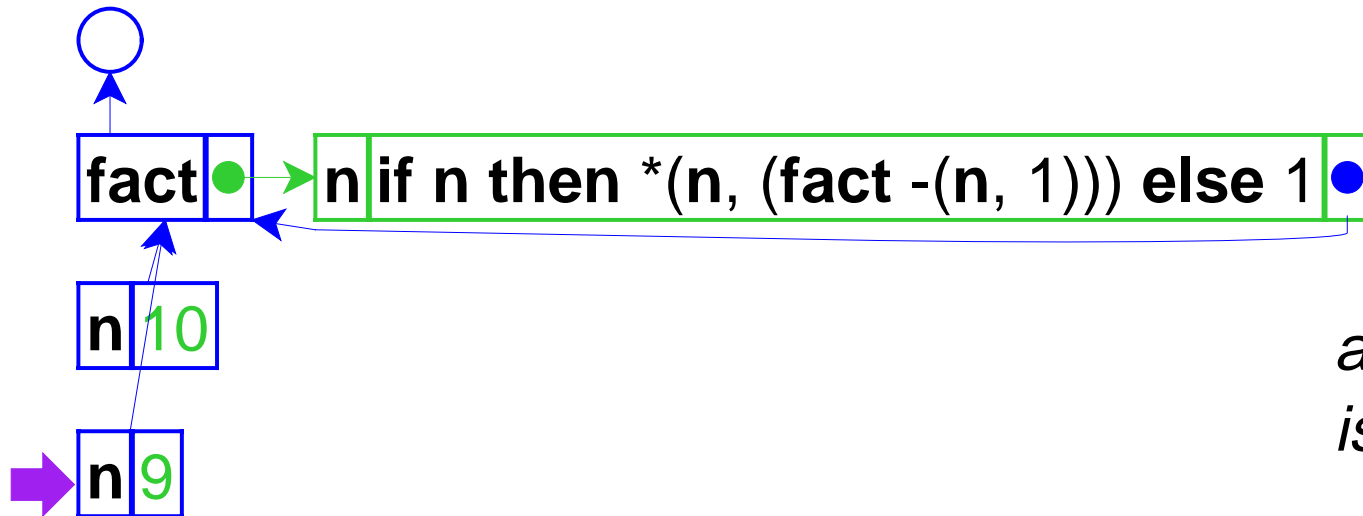
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
in (fact 10)
```

## 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



*an advantage: closure  
is only created once*

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

# 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 vec1743 (vector 1 2 3))
```

```
(let ([v vec1743]) (vector-ref v 0))
```

→

```
... (define vec1743 (vector 1 2 3))
```

```
(vector-ref vec1743 0)
```

→

```
... (define vec1743 (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 vec1743 (vector 1 2 3))
```

```
(let ([v vec1743]) (begin (vector-set! v 0 5) (vector-ref v 0)))
```

→

```
... (define vec1743 (vector 1 2 3))
```

```
(begin (vector-set! vec1743 0 5) (vector-ref vec1743 0))
```

→

```
... (define vec1743 (vector 5 2 3))
```

```
(vector-ref vec1743 0)
```

→

```
... (define vec1743 (vector 5 2 3))
```

5

# Begin Expressions

- **begin** evaluates a sequence of expressions, in order
- **lambda** and **let** always supply an implicit **begin**

**(let (...)**  $\langle\text{expr}\rangle_1$  ...  $\langle\text{expr}\rangle_n$ )  
= **(let (...)** (**begin**  $\langle\text{expr}\rangle_1$   $\langle\text{expr}\rangle_n$ ))

**(lambda (...)**  $\langle\text{expr}\rangle_1$  ...  $\langle\text{expr}\rangle_n$ )  
= **(lambda (...)** (**begin**  $\langle\text{expr}\rangle_1$   $\langle\text{expr}\rangle_n$ ))

# Changing Recursive Environment Extension

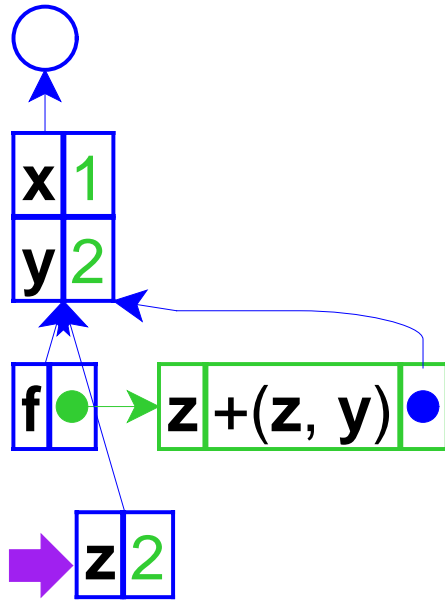
Now we can change `extend-env-recursively` to use **vector-set!**

Go back to just two datatype variants

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

(implement in DrScheme)

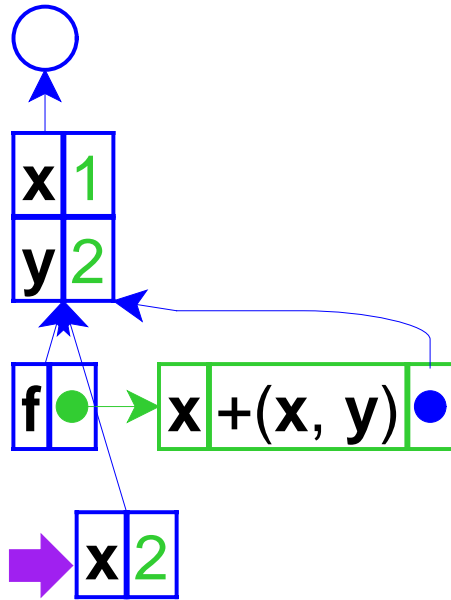
# Back to Lexical Scope



What if we change `z` to `x`?

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

## Back to Lexical Scope

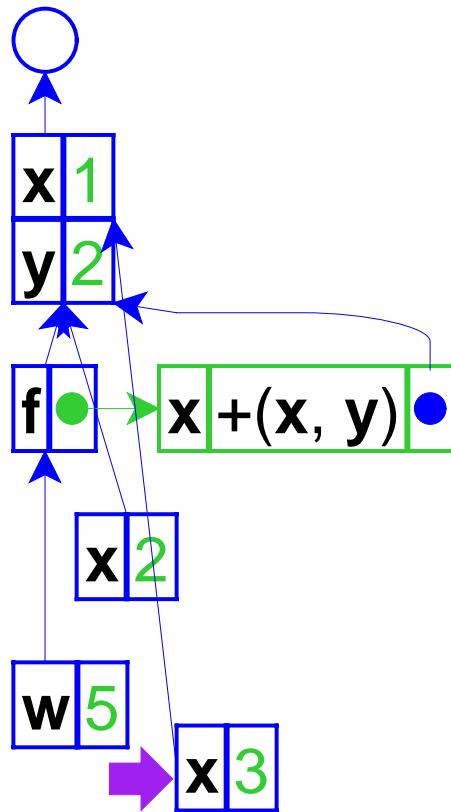


Shape of the environment and location of the argument is unchanged

- 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)
```

# Back to Lexical Scope



Still true if `f` is called from a more complex environment

```
let x = 1 y = 2
in let f = proc (x) +(x, y)
in +((f y), let w = 5 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

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

to

```
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