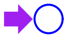
Environments in Picture Form

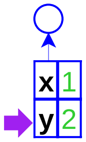


top purple arrow points to the current environment

purple in bottom area hilites the current expression

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

Environments in Picture Form

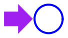


top purple arrow points to the current environment

purple in bottom area hilites the current expression

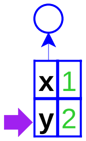
```
let x = 1  y = 2
  in +(x, y)
```

Environments in Picture Form



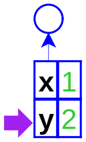
```
let x = 1  y = 2
  in let f = proc (z) +(z, y)
    in (f y)
```

Environments in Picture Form



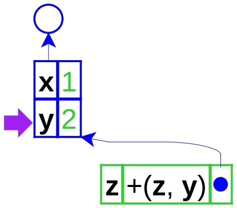
```
let x = 1  y = 2
  in let f = proc (z) +(z, y)
    in (f y)
```

Environments in Picture Form



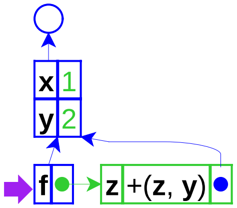
```
let x = 1  y = 2
  in let f = proc (z) +(z, y)
    in (f y)
```

Environments in Picture Form



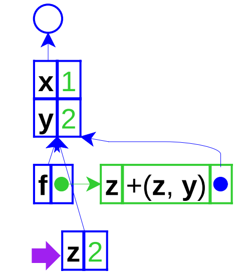
```
let x = 1  y = 2
  in let f = proc (z) +(z, y)
    in (f y)
```

Environments in Picture Form



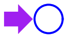
```
let x = 1  y = 2
  in let f = proc (z) +(z, y)
    in (f y)
```

Environments in Picture Form



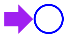
```
let x = 1  y = 2
  in let f = proc (z) +(z, y)
    in (f y)
```

The Need for Recursive Environments



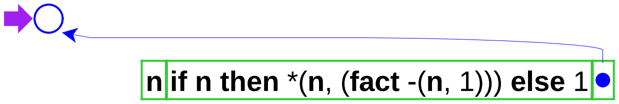
```
let fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
 in (fact  10)
```

The Need for Recursive Environments



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

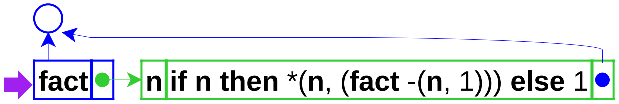
The Need for Recursive Environments



n | if n then *(n, (fact -(n, 1))) else 1

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

The Need for Recursive Environments

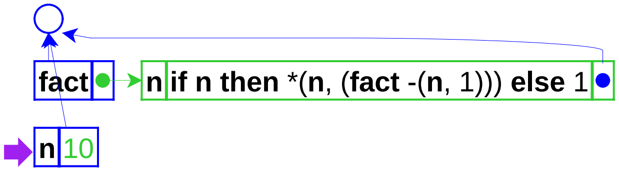


fact

n | if n then *(n, (fact -(n, 1))) else 1

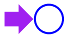
```
let fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
 in (fact  10)
```

The Need for Recursive Environments



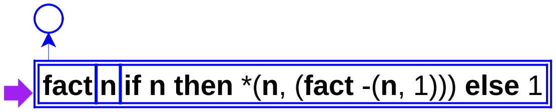
```
let fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
 in (fact  10)
```

The Need for Recursive Environments



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

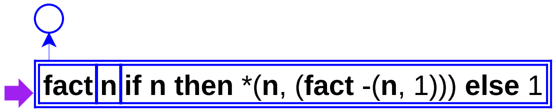
The Need for Recursive Environments



double box means a recursively extended environment

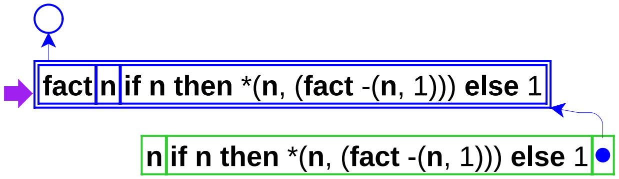
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
 in (fact  10)
```

The Need for Recursive Environments



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

The Need for Recursive Environments



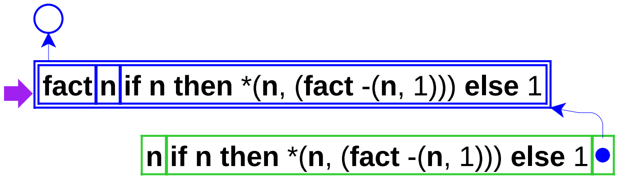
fact | n | if n then *(n, (fact -(n, 1))) else 1

n | if n then *(n, (fact -(n, 1))) else 1 | ●

every lookup of **fact**
generates a closure

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

The Need for Recursive Environments

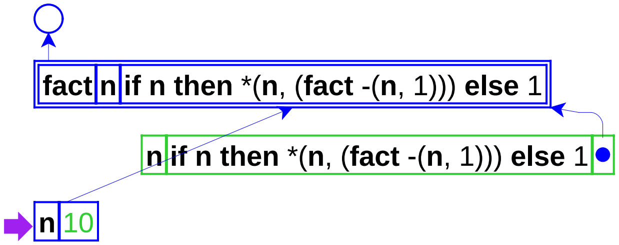


fact | n | if n then *(n, (fact -(n, 1))) else 1

n | if n then *(n, (fact -(n, 1))) else 1 | ●

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

The Need for Recursive Environments



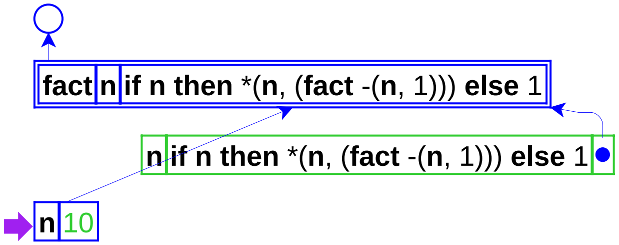
fact | n | if n then *(n, (fact -(n, 1))) else 1

n | if n then *(n, (fact -(n, 1))) else 1 | ●

n | 10

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

The Need for Recursive Environments



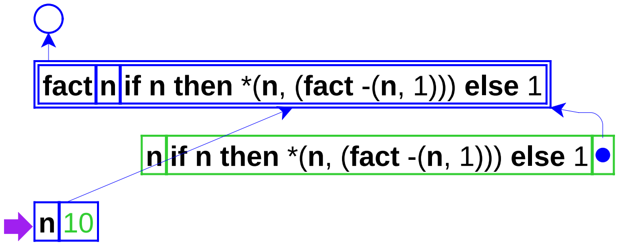
fact | n | if n then *(n, (fact -(n, 1))) else 1

n | if n then *(n, (fact -(n, 1))) else 1 | ●

n | 10

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

The Need for Recursive Environments



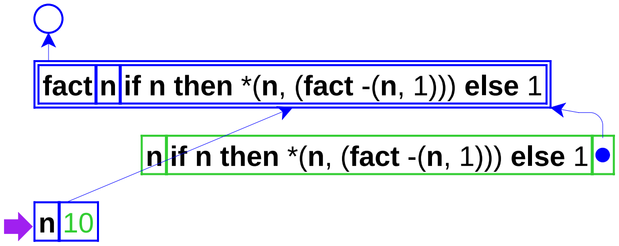
fact | n | if n then *(n, (fact -(n, 1))) else 1

n | if n then *(n, (fact -(n, 1))) else 1 | ●

n | 10

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

The Need for Recursive Environments



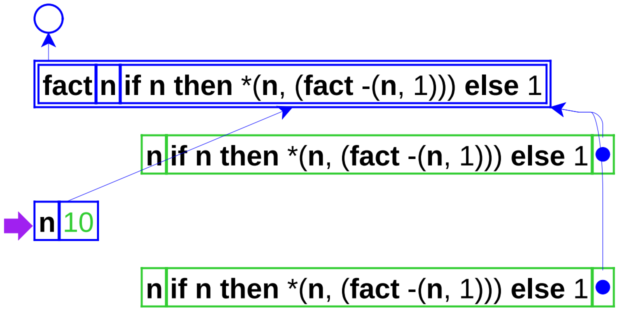
fact | n | if n then *(n, (fact -(n, 1))) else 1

n | if n then *(n, (fact -(n, 1))) else 1 | ●

n | 10

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

The Need for Recursive Environments



fact | n | if n then *(n, (fact -(n, 1))) else 1

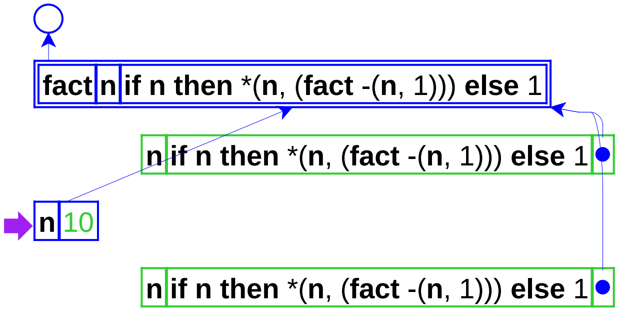
n | if n then *(n, (fact -(n, 1))) else 1 | ●

n | 10

n | if n then *(n, (fact -(n, 1))) else 1 | ●

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

The Need for Recursive Environments



fact | n | if n then *(n, (fact -(n, 1))) else 1

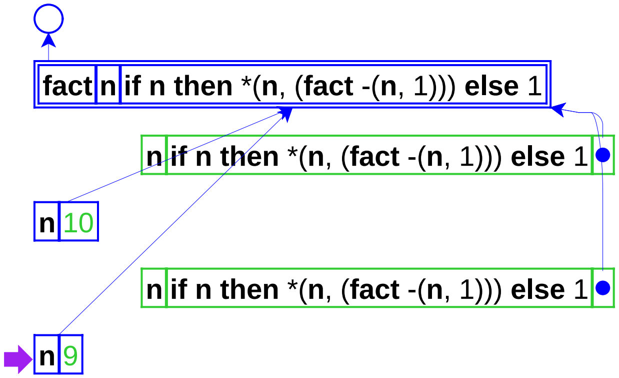
n | if n then *(n, (fact -(n, 1))) else 1 | ●

n | 10

n | if n then *(n, (fact -(n, 1))) else 1 | ●

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

The Need for Recursive Environments



fact | n | if n then *(n, (fact -(n, 1))) else 1

n | if n then *(n, (fact -(n, 1))) else 1 | ●

n | 10

n | if n then *(n, (fact -(n, 1))) else 1 | ●

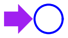
n | 9

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

Implementing Recursively Extended Envs

(implement in DrScheme)

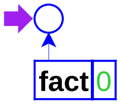
Another Approach to Recursive Closures



alternate approach...

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

Another Approach to Recursive Closures

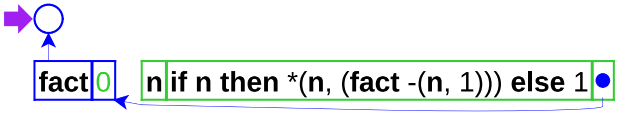


fact | 0

create an environment with a dummy value...

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

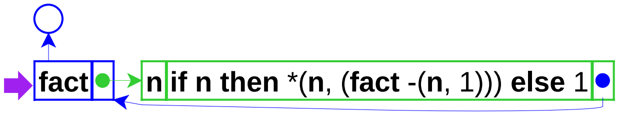
Another Approach to Recursive Closures



create the closure using the environment...

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

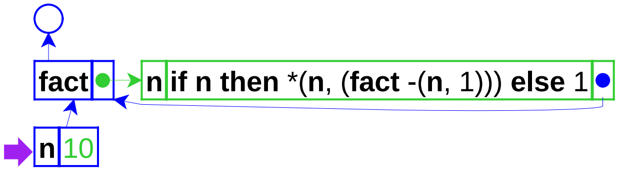
Another Approach to Recursive Closures



then
modify
the environment
to fix it up

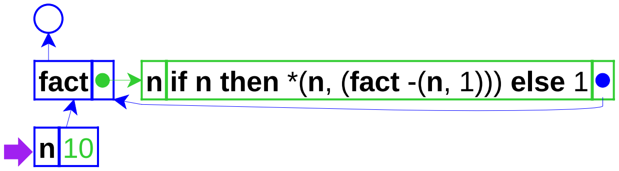
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
 in (fact  10)
```

Another Approach to Recursive Closures



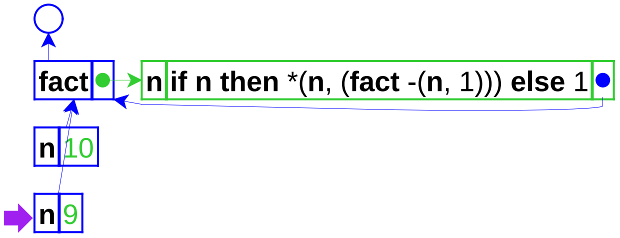
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
 in (fact  10)
```

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


fact

n if n then *(n, (fact -(n, 1))) else 1

n 10

n 9

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

- The part of the environment that we need to modify is a value in a vector
- So we need evaluation rules to support 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**

(**let** (...) <expr>$_1$ **...** <expr>$_n$)
= (**let** (...) (**begin** <expr>$_1$ <expr>$_n$))

(**lambda** (...) <expr>$_1$ **...** <expr>$_n$)
= (**lambda** (...) (**begin** <expr>$_1$ <expr>$_n$))

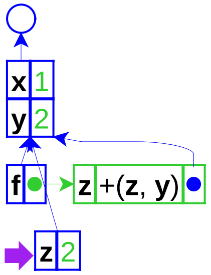
Changing Recursive Environment Extension

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

Go back to just two datatype variants

(implement in DrScheme)

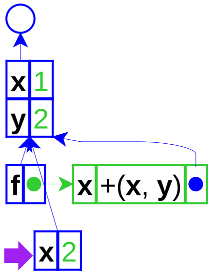
Back to Lexical Scope



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

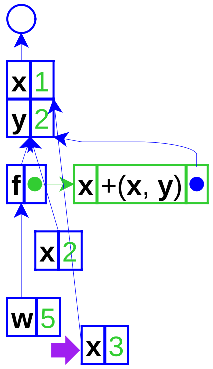


Shape of the environment and location of the argument is unchanged

- 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


x 1
y 2
f
x +(x, y)
x 2
w 5
x 3

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