Terminology: Denoted and Expressed Values

- A ***denoted value*** is the meaning of a variable
- An ***expressed value*** is the result of an expression

The set of denoted and expressed values can be different

Terminology: Denoted and Expressed Values

- First-order functions
  - denoted values: numbers and functions
  - expressed values: numbers
- Higher-order functions
  - denoted values: numbers and functions
  - expressed values: numbers and functons

Procedure Expressions: Concrete Syntax

<prog> ::= <expr>

<expr> ::= **proc** (<id>$^{*(,)}$) <expr>

::= (<expr> <expr>*)

let identity = **proc**(**x**) **x**

in (**identity** 5)

Procedure Expressions: Abstract Syntax

```
<prog>    ::=   (a-program <expr>)
<expr>    ::=   (proc-exp (list <id>*) <expr>)
          ::=   (app-exp <expr> (list <expr>*))
<val>     ::=   <num>   |   <proc>
<proc>    ::=   (closure (list <id>*) <expr> <env>)
```

```
(a-program
  (let-exp (list 'identity)
           (list (proc-exp (list 'x) (var-exp 'x)))
           (app-exp (var-exp 'identity) (list-exp 5))))
```

Implementing Procedures

(implementation in DrScheme)

New representation of environments:

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

Recursion

Suppose we try to write the **fact** function using only **let**

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

The above doesn't work, because **fact** is not bound in the local function

We'll add **letrec**, but first we'll see how to implement **fact** without it...

Recursion with Let

- Problem: **fact** can't see itself
- Note: anyone calling **fact** can see **fact**
- Idea: have the caller supply **fact** to **fact** (along with a number)

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

this works!

What Happened?

- The key insight is delaying some work to the caller
- We can exploit this idea to implement **letrec**, but in a slightly different way

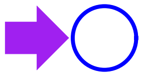
- **letrec** requires a *closure* that refers to itself
- We can delay the actual construction of the closure until it is extracted from the environment

Recursive Environments for Recursive Functions

This isn't going to work

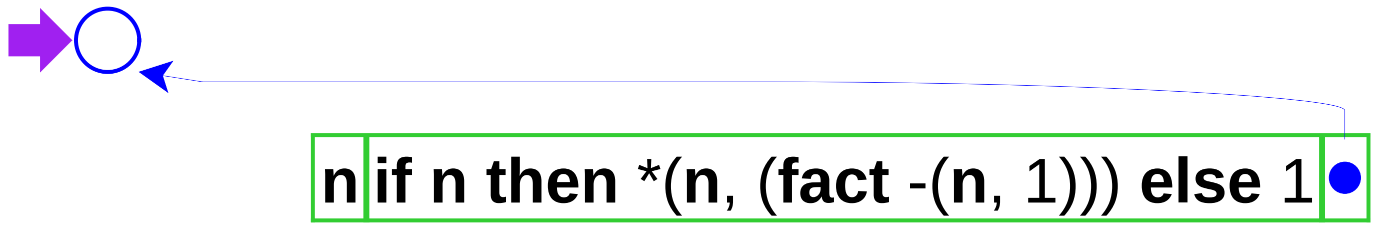
```
let fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
 in (fact  10)
```

Recursive Environments for Recursive Functions



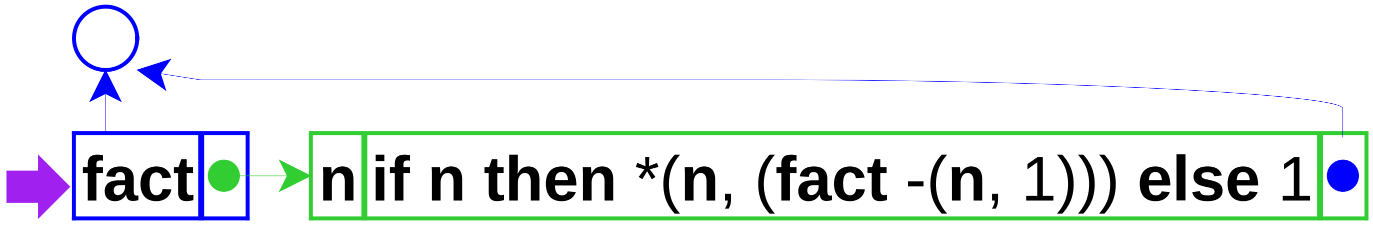
```
let fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
 in (fact  10)
```

Recursive Environments for Recursive Functions



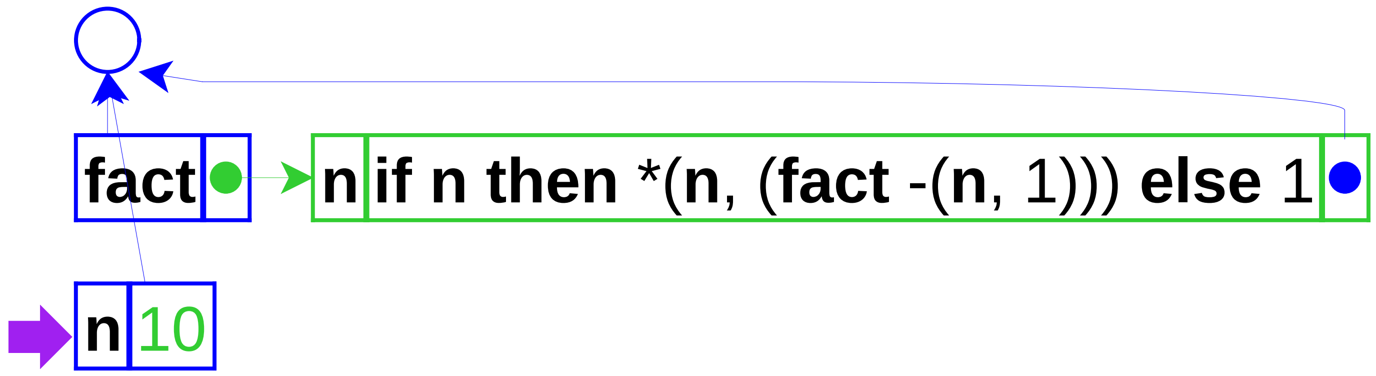
```
let fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
 in (fact  10)
```

Recursive Environments for Recursive Functions



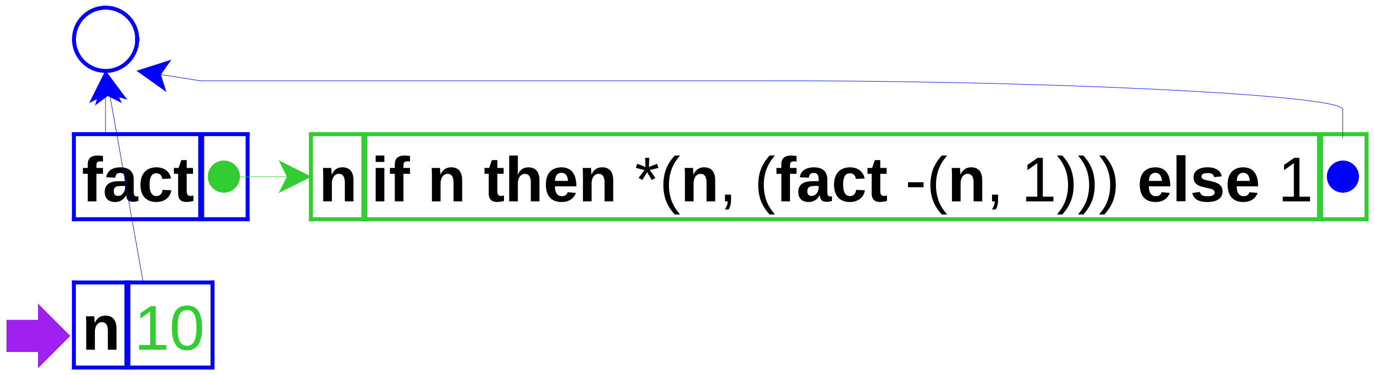
let fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
 in (**fact** 10)

Recursive Environments for Recursive Functions



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

Recursive Environments for Recursive Functions



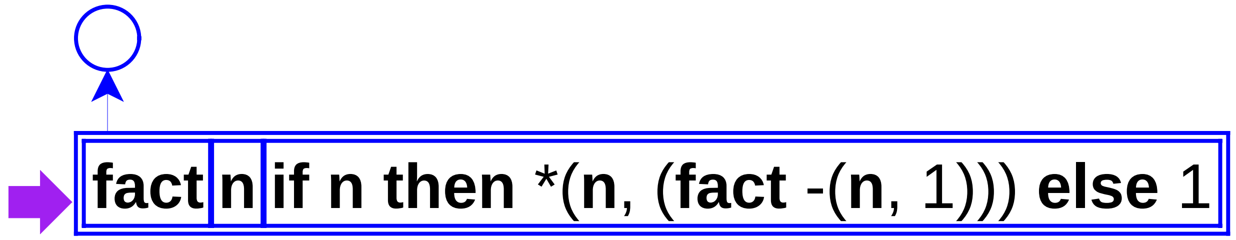
No binding for **fact**

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

Recursive Environments for Recursive Functions

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

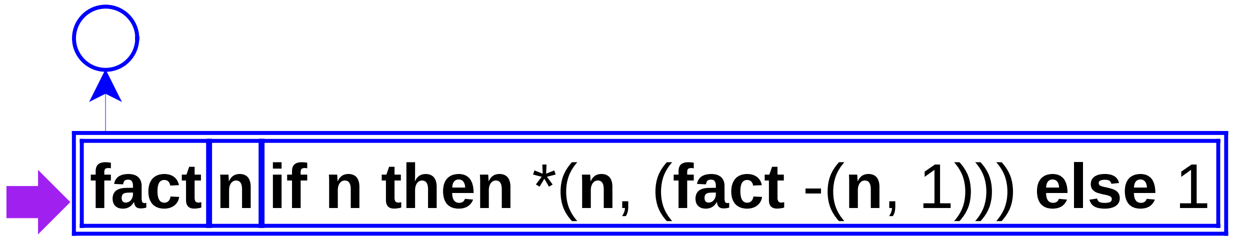
Recursive Environments for Recursive Functions



double box means a recursively extended environment

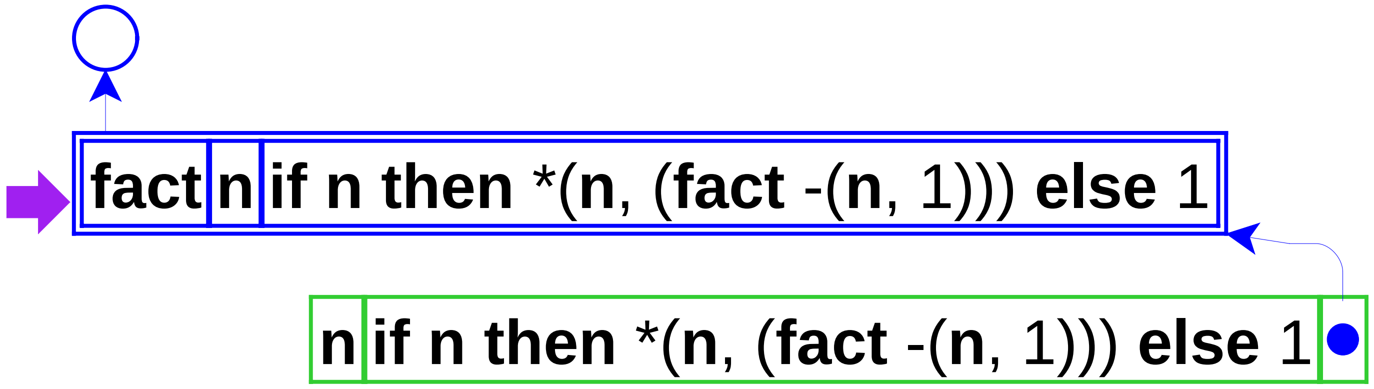
letrec fact = **proc**(**n**) **if n then** *(**n**, (**fact** -(**n**, 1))) **else** 1
 in (fact 10)

Recursive Environments for Recursive Functions



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

Recursive Environments for Recursive Functions

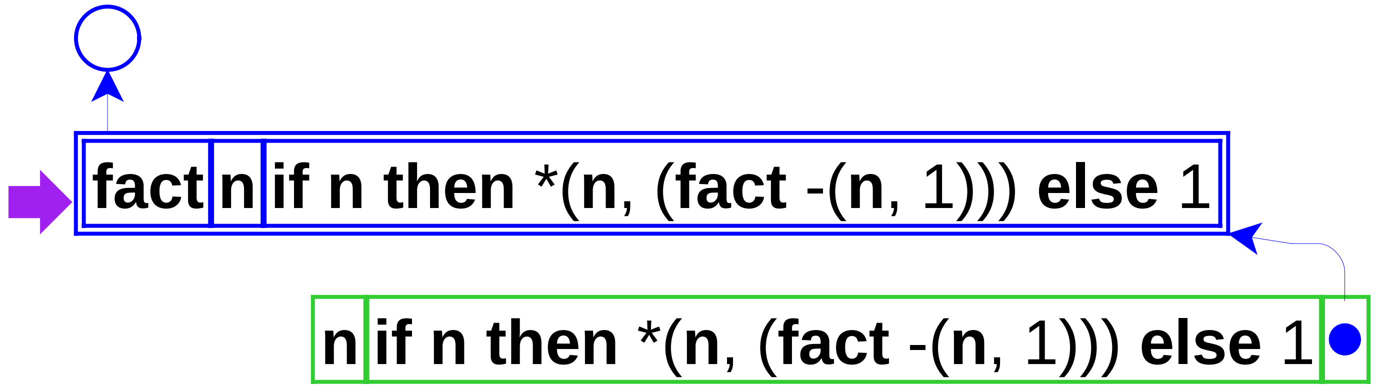


fact
n
if n then *(n, (fact -(n, 1))) else 1
n
if n then *(n, (fact -(n, 1))) else 1

every lookup of **fact**
generates a closure

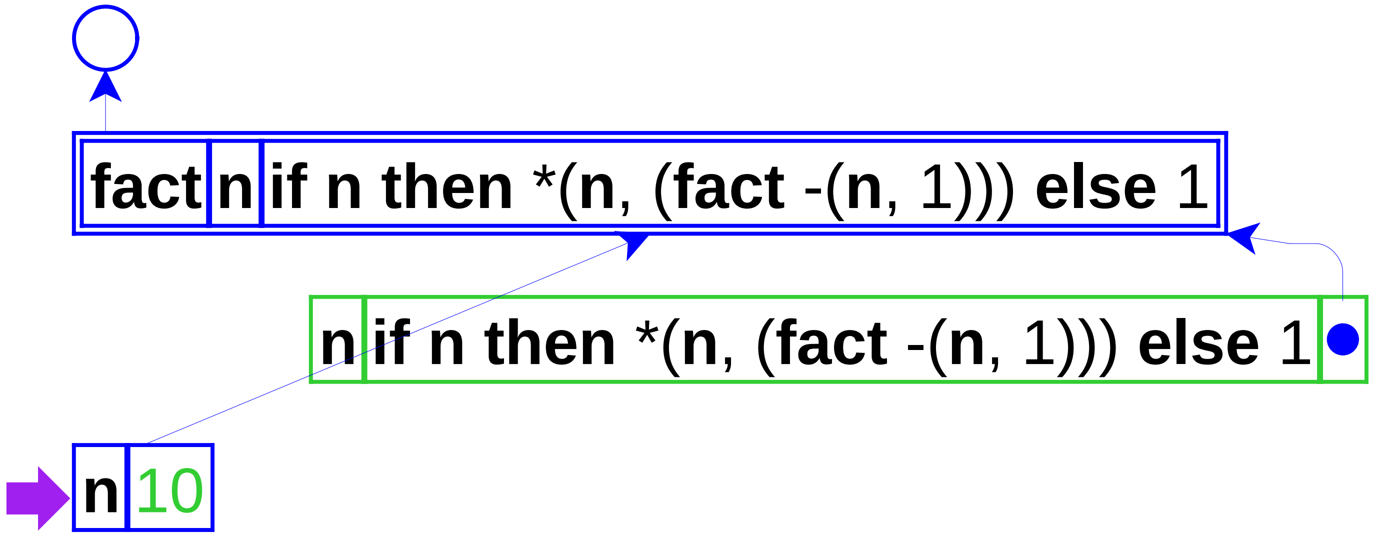
letrec fact = **proc**(**n**) **if n then** *(**n**, (**fact** -(**n**, 1))) **else** 1
 in (**fact** 10)

Recursive Environments for Recursive Functions


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

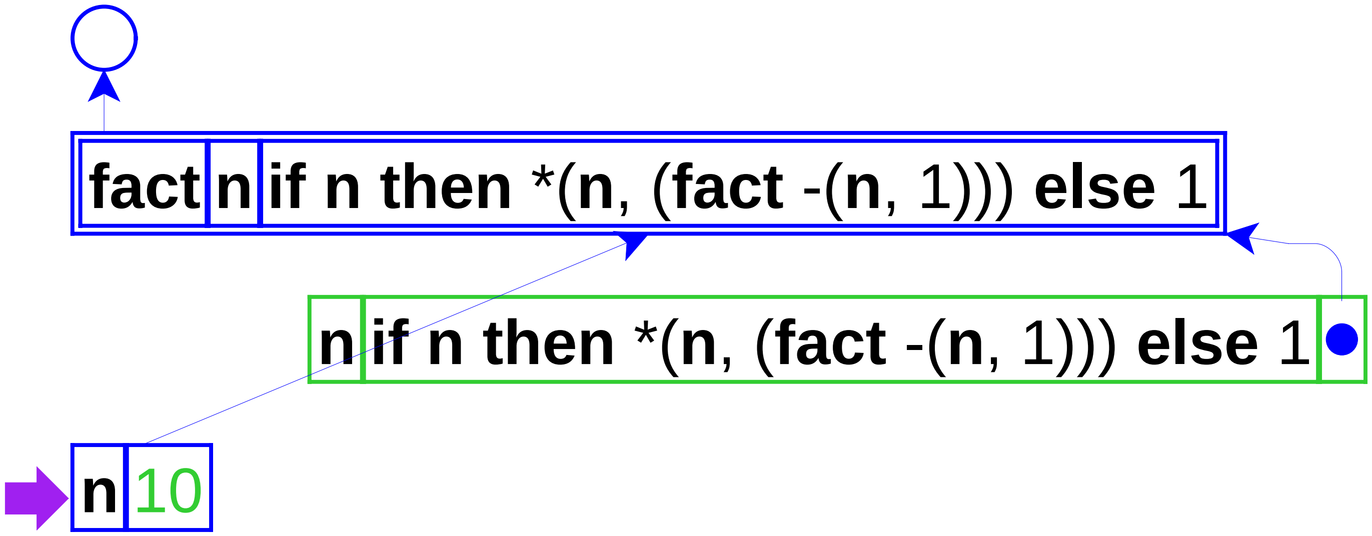
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
 in (fact  10)
```

Recursive Environments for Recursive Functions


fact
n
if n then *(n, (fact -(n, 1))) else 1
n
if n then *(n, (fact -(n, 1))) else 1
n
10

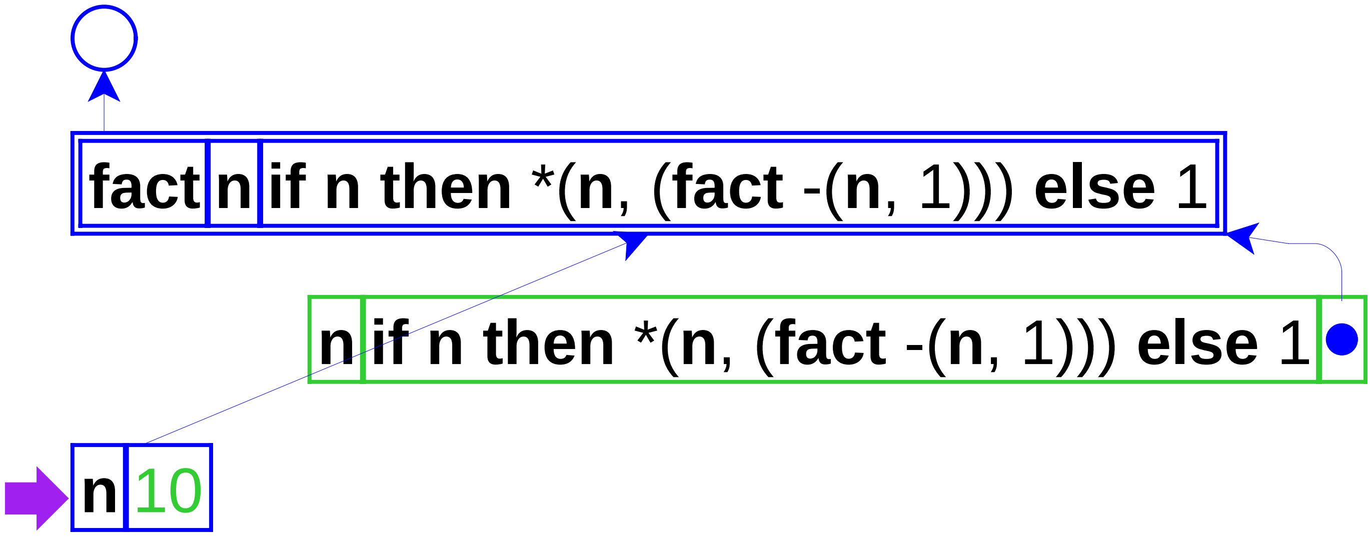
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
 in (fact  10)
```

Recursive Environments for Recursive Functions


fact
n
if n then *(n, (fact -(n, 1))) else 1
n
if n then *(n, (fact -(n, 1))) else 1
n
10

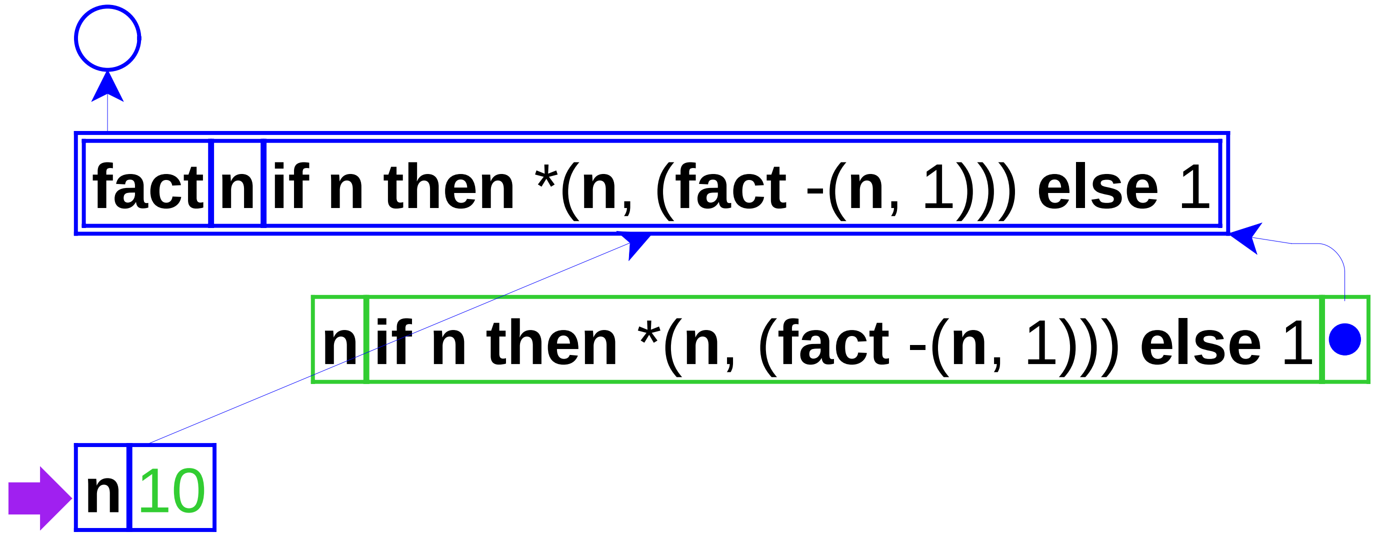
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
 in (fact  10)
```

Recursive Environments for Recursive Functions


fact
n
if n then *(n, (fact -(n, 1))) else 1
n
if n then *(n, (fact -(n, 1))) else 1
n
10

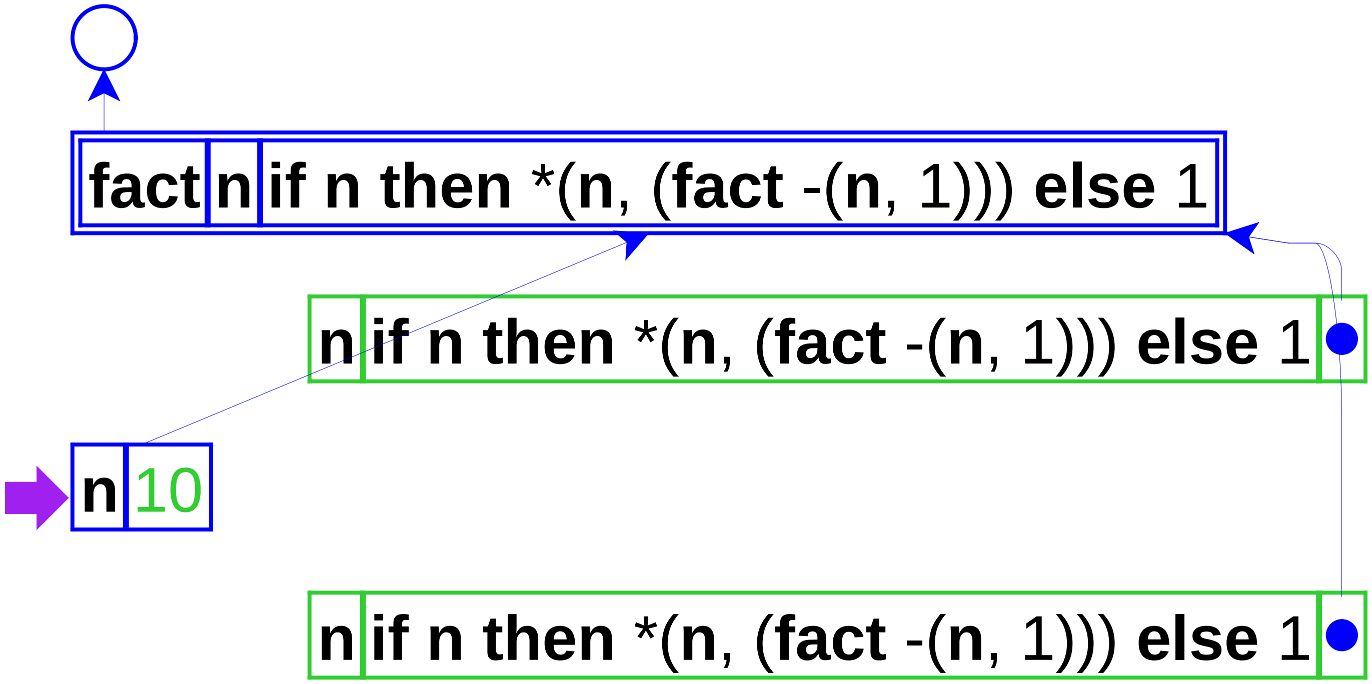
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
 in (fact  10)
```

Recursive Environments for Recursive Functions


fact
n
if n then *(n, (fact -(n, 1))) else 1
n
if n then *(n, (fact -(n, 1))) else 1
n
10

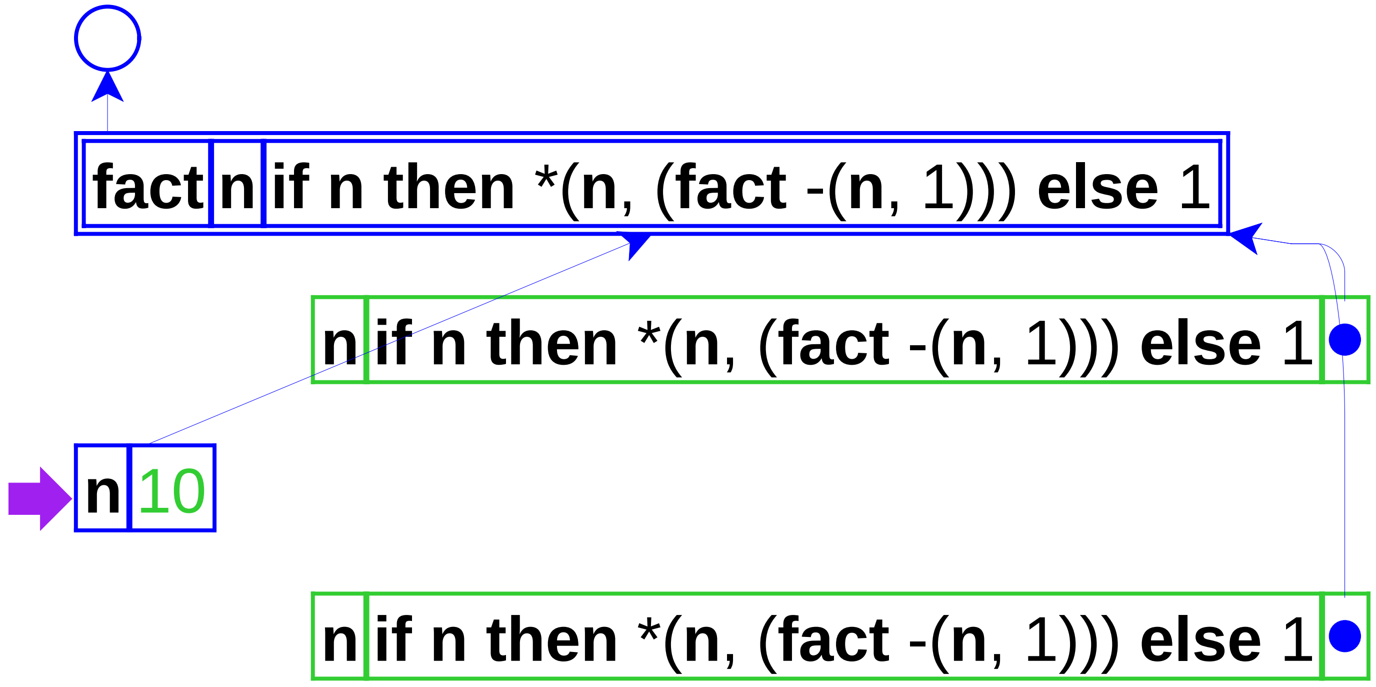
letrec fact = **proc**(**n**) **if n then** *(**n**, (**fact** -(**n**, 1))) **else** 1
 in (**fact** 10)

Recursive Environments for Recursive Functions


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

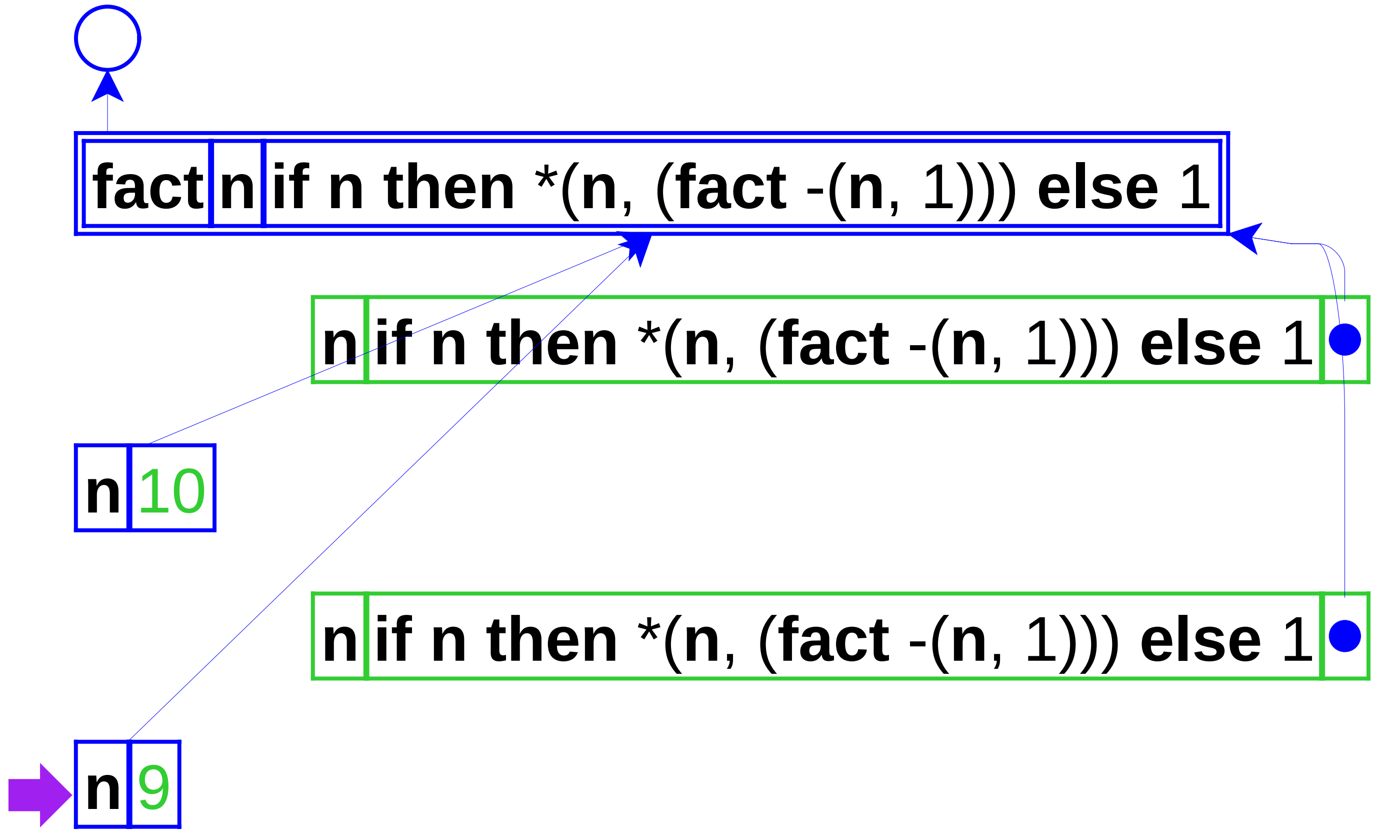
```
letrec fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
 in (fact  10)
```

Recursive Environments for Recursive Functions


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)

Recursive Environments for Recursive Functions


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

```
(define-datatype environment environment?
  (empty-env-record)
  (extended-env-record
    (syms (list-of symbol?))
    (vals (list-of denval?))
    (env environment?))
  (recursively-extended-env-record
    (proc-names (list-of symbol?))
    (idss (list-of (list-of symbol?)))
    (bodies (list-of expression?))
    (env environment?)))
```

Implementing letrec

(implement in DrScheme)

Back to Recursion with Let...

- Allowing functions to be values is a powerful idea
- As it turns out, we don't even need **let**!

let <id>$_1$ = <expr>$_1$ **...** <id>$_n$ = <expr>$_n$ **in** <expr>

is the same as

(**proc**(<id>$_1$, **...** <id>$_n$) <expr> <expr>$_1$ **...** <expr>$_n$)

Back to Recursion with Let...

- Allowing functions to be values is a powerful idea
- As it turns out, we don't even need **let**!

(**let** ([<id>$_1$ <expr>$_1$] **...** [<id>$_n$ = <expr>$_n$]) <expr>)

is the same as

((**lambda** (<id>$_1$ **...** <id>$_n$) <expr>) <expr>$_1$ **...** <expr>$_n$)

The Lambda Calculus

- We don't even need functions of multiple arguments...

((**lambda** (<id>$_1$ **...** <id>$_n$) <expr>)
<expr>$_1$ **...** <expr>$_n$)

is the same as

(((**lambda** (<id>$_1$) **...** (**lambda** (<id>$_n$) <expr>))
<expr>$_1$) **...**
<expr>$_n$)

Passing multiple arguments one-at-a-time is called ***currying***

The ***lambda calculus*** has only single-argument **lambda** and single-argument function calls, and it's computationally complete