Procedures

(finish implementation in DrScheme)

Different representation of environments:

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

Recusion

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...

Recusion 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 an *environment* that refers to itself
- We can delay the actual construction of the enviornment until the environment is used

Recursive Environments

```
(define-datatype environment environment?
  (empty-env-record)
  (extended-env-record
    (syms (list-of symbol?))
    (vec vector?)
    (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 Recusion with Let: What Really Happened?

- 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 Recusion with Let: What Really Happened?

- 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