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 functions
Procedure Expressions: Concrete Syntax

\[
\text{<prog>} ::= \text{<expr>}
\]
\[
\text{<expr>} ::= \text{proc (}<id>^{*(),}) \text{<expr>}
\]
\[
::= (\text{<expr> <expr>}^{*})
\]

let identity = \text{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 \texttt{fact} function using only \texttt{let}

\begin{verbatim}
let fact = proc(n) if n then *(n, (fact -(n, 1))) else 1
in (fact 10)
\end{verbatim}

The above doesn't work, because \texttt{fact} is not bound in the local function

We'll add \texttt{letrec}, but first we'll see how to implement \texttt{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)

\[
\text{let } \text{fact} = \text{proc}(n, f) \text{ if } n \text{ then } *(n, (f - (n, 1) f)) \text{ else } 1 \\
\text{in } (\text{fact } 10 \text{ fact})
\]

\text{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

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

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

No binding for fact
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

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

double box means a recursively extended environment
Recursive Environments for Recursive Functions

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

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

every lookup of **fact** generates a closure
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

\[ \text{letrec fact = proc(n) } \begin{cases} \text{if } n \text{ then } *(n, \text{(fact } -(n, 1))) \text{ else } 1 \\ \text{in (fact 10)} \end{cases} \]
Recursive Environments for Recursive Functions

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

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

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

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

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

```
n 10
```

```
n 9
```
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 \textbf{let}!

\[
\text{let } <id>_1 = <expr>_1 \ldots <id>_n = <expr>_n \text{ in } <expr>
\]

is the same as

\[
(\text{proc}(<id>_1, \ldots <id>_n) <expr> <expr>_1 \ldots <expr>_n)
\]
• Allowing functions to be values is a powerful idea

• As it turns out, we don't even need let!

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

is the same as

\[
((\text{lambda} \ (<\text{id}>_1 \ ... \ <\text{id}>_n) \ <\text{expr}>) \ <\text{expr}>_1 \ ... \ <\text{expr}>_n)
\]
The Lambda Calculus

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

  $$\left(\left(\text{lambda} \ (<id>_1 \ldots <id>_n) <\text{expr}>\right)\right)$$

  $$<\text{expr}>_1 \ldots <\text{expr}>_n)$$

  is the same as

  $$\left(\left(\text{lambda} \ (<id>_1) \ldots (\text{lambda} \ (<id>_n) <\text{expr}>))\right)\right)$$

  $$<\text{expr}>_1) \ldots$$

  $$<\text{expr}>_n)$$

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