RACKET

Let Expressions

Let Expressions

- A let expressions enables us to define local variables.
- Examples:

```
(let ([x 3]) x)

(let ([x 3] [y 4]) (+ x y))

(define pythagoras
  (lambda (x y)
    (let ([x-squared (* x x)]
          [y-squared (* y y)])
      (sqrt (+ x-squared y-squared)))))
```

Let Expressions

- Let expressions are equivalent to Lambda-Application expressions.

```
(let ([x 3]) x)
⇔ ((lambda (x) x) 3)

(let ([x 3] [y 4]) (+ x y))
⇔ ((lambda (x y) (+ x y)) 3 4)
```

- Class exercise: convert pythagoras

Let Expressions

```
(define pythagoras
  (lambda (x y)
    (let ([x-squared (* x x)]
          [y-squared (* y y)])
     (sqrt (+ x-squared y-squared))])))

(define pythagoras
  (lambda (x y)
     ((lambda (x-squared y-squared)
         (sqrt (+ x-squared y-squared)))
       (* x x)
       (* y y))))
```

Let Expressions

```
(define hmm
   (lambda (l)
     (let ([a  (car l)]
           [b  (cdr l)]
           [c  (car b)])
       (list c a))))
```

What happens when we evaluate `(hmm '(1 2 3))`?

Translate the `let` expression to an application of `lambda`

Let* Expressions

```
(define hmm
   (lambda (l)
     (let* ([a  (car l)]
            [b  (cdr l)]
            [c  (car b)])
        (list c a))))
```

This is translated to:

```
(define hmm
   (lambda (l)
     (let ([a  (car l)])
       (let ([b  (cdr l)])
         (let ([c  (car b)])
           (list c a))))))
```

Information Hiding – Local Procedures

- Sometimes, local procedures can be helpful.
- Everything we need is right there.

```
(define fact
  (lambda (n)
    (let ([fact-tail (lambda (n accu)
                       (if (zero? n)
                           accu
                           (fact-tail (- n 1)
                                      (* n accu))))])
    (fact-tail n 1))))
```

- The code above is perfectly fine, except for the use of a let.
- Can you explain why the “let” does not work?

Information Hiding – Local Recursive Procedures

- To fix the problem with the “let” we use “letrec:”

```
(define fact
  (lambda (n)
    (letrec ([fact-tail (lambda (n accu)
                       (if (zero? n)
                           accu
                           (fact-tail (- n 1)
                                      (* n accu))))])
    (fact-tail n 1))))
```

Information Hiding – Local Recursive Procedures

- An alternate way of writing the procedure is:

```
(define fact
  (letrec ([fact-tail (lambda (n accu)
                        (if (zero? n)
                            accu
                            (fact-tail (- n 1)
                                       (* n accu))))])
    (lambda (n)(fact-tail n 1))))
```

- Here is the code from the prior slide:

```
(define fact
  (lambda (n)
    (letrec ([fact-tail (lambda (n accu)
                        (if (zero? n)
                            accu
                            (fact-tail (- n 1)
                                       (* n accu))))])

 (fact-tail n 1))))
```

- What might be the benefit of the "new and improved" code?

Information Hiding – Local Recursive Procedures

- Another example of local procedures.

```
(define odd?
  (letrec ([odd? (lambda (n)
                   (if (zero? n)
                       #f
                       (even? (- n 1))))]
           [even? (lambda (n)
                    (if (zero? n)
                        #t
                        (odd? (- n 1))))])
    (lambda (n)
      (odd? n))))
```

- Mutual recursion.

Named Let, aka, turning your mind into a pretzel

- Consider:

```
(define fact
  (lambda (n)
      (let foo ([x n] [accu 1])
         (if (zero? x)
              accu
              (foo (- x 1)
                   (* accu x))))))
```

- Does this work?
- If so, why?
- If not, why not?

Parameters

- Consider:

```
((lambda (x y) (cons x y)) 3 '())
```

- We know that 3 gets assigned to x and '() to y.
- Now consider:

```
((lambda l l) '3)      -> (3)
```

- The value 3 gets assigned to l, but places it in a list.
- This comes in handy when we have procedures with a variable number of parameters.
- We can do:

```
((lambda l l) '3 '3)  -> (3 3)
```

Parameters

- To force a minimum number of arguments use improper lists.

```
((lambda (x y . z) (list x y z)) 3 4)
      -> (3 4 ())

((lambda (x y . z) (list x y z)) 3 4 5)
      -> (3 4 (5))
```