RACKET

1st Class Procedures

Procedures returning procedures

- In Racket, we can write procedures which in turn return procedures as values.
- Because of this, we say that in Racket, we have 1st class procedures.
- Example:

```
(lambda(x) (lambda (y) (+ y 1)))
```

- The above (outer) procedure returns a procedure which adds one to the argument passed to it.
- We invoke the who expression like so:

```
((lambda(x) (lambda (y) (+ y 1))) 5)
```

A Slightly More Complex Example

- Let's consider:

```
(((lambda (x) (lambda (y) (+ x y))) 3) 4)
```

- At the top level, we have two elements, indicated in green brackets and a green number:

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

- Since the top level expression is an application, the expression in brackets needs to be evaluated:

```
[(lambda (x) (lambda (y) (+ x y))) 3]
```

A Slightly More Complex Example

```
[(lambda (x) (lambda (y) (+ x y))) 3]
```

- It itself has two elements, indicated in red brackets and a red number:

```
([lambda (x) (lambda (y) (+ x y))] 3)
```

- Just as before, we need to evaluate the expression in the brackets.
- It results in a procedure with the parameter x.
- Next, we apply this procedure to 3:
- This results in an assignment of 3 to x.

- We then evaluate the body of this procedure.

A Slightly More Complex Example

- We then evaluate the body of this procedure, i.e.:

```
(lambda (y) (+ x y))
```

- The value of the body is a procedure with the parameter y.
- This procedure is returned to replace the “green” expression.
- In other words, the followin expression from a few slides ago:

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

- Becomes:

```
([lambda (y) (+ x y)] 4)
```

A Slightly More Complex Example

```
([lambda (y) (+ x y)] 4)
```

- Next, we assign 4 to y.

- Next, we evaluate the body of this procedure:

```
(+ x y)
```

- Looking ar the variable assignments for x and y, Scheme returns 7.

Procedures returning procedures

- Class exercise:. What happens when we execute?

```
(((lambda(x) x) (lambda (y) y)) 5)
```

Shadowing

- What would the value of the expression be, if we had instead:

```
(((lambda (x) (lambda (x) (+ x x))) 3) 4)
```

A More Intuitive Example

- Consider:

```
(define add-generator
  (lambda (n)
    (lambda (m)
      (+ n m))))
```

- Add-generator is a procedure which generates procedures.
- The procedure colored green is returned to us.
- Notice that whatever we pass in as `n` is locked in.
- Let's generate a procedure which adds 42 to the argument passed to it:

```
(add-generator 42)
```

- It returns a procedure like so:

```
(lambda (m)
   (+ 42 m))
```

A More Intuitive Example

- Let's save the procedure returned to us:

```
(define add42
 (add-generator 42))
```

- We can now invoke it like so:

```
(add42 7)    -> 7
```

Curried Procedures

- A procedure of n arguments can be converted into a set of n procedures of one argument each.
- Example.

```
(define regular-cons
  (lambda (h t)
    (cons h t)))
```

- Converted to a curried procedure:

```
(define curry-cons
   (lambda (h)
      (lambda (t)
         (cons h t))))
```