MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1999

Recitation – Friday, February 5

1. Practice with Scheme

Write down what Scheme will print in response to the following:

```
⇒ (define four 4)
;Value: "four --> 4"
⇒ four
;Value: 4
⇒ (define six (lambda () 6))
;Value: "six --> #[compound procedure]"
⇒ six
;Value: #[compound procedure]
⇒ (+ four 1)
;Value: 5
⇒ (+ (four) 1)
;Error: The object 4 is not applicable
⇒ (+ six 1)
;Error: Compound procedure passed to add
⇒ (+ (six) 1)
;Value: 7
⇒ (define f-add
    (lambda (x y)
      (+ (x) (y))))
;Value: "f-add --> #[compound procedure]"
⇒ (f-add six (lambda () four))
;Value: 10
```

```
⇒ (define a 5)
;Value: "a --> 5"
⇒ (define b (+ 1 a))
;Value: "b --> 6"
⇒ (+ 2 (if (> b a) b a))
;Value: 8
⇒ (if #t 5 (/ 1 0))
;Value: 5
⇒ ((if (< a b) + -) a b)
;Value: 11
⇒ (if (if + #f 2) 3 6)
;Value: 6
⇒ (if 0 1 2)
;Value: 1
⇒ (define n 4)
;Value: "n --> 4"
⇒ (define increment (lambda (x) (+ x 1)))
;Value: "increment --> #[compound proc]"
⇒ (increment n)
;Value: 5
⇒ n
;Value: 4
```

2. Writing Simple Procedures

Write a procedure `max` of two arguments that returns the larger one.

```
(define max
  (lambda (x y)
    (if (> x y)
        x
        y)))
```

Write a procedure `sign` that takes a number as its argument and returns -1 if it is negative, 1 if it is positive and 0 if the argument is zero.

```
(define sign
  (lambda (x)
    (if (> x 0)
        1
        (if (< x 0)
            -1
            0))))
```

3. Scoping of Variables

Consider the example below. Notice that `x` is used in multiple places. When do we substitute for `x` and when don't we?

```
⇒ (define x-y*y
    (lambda (x y)
      (- x ((lambda (x) (* x x)) y))))
```

Use the substitution model to evaluate the following expression:

```
⇒ (x-y*y 11 3)
⇒ ([proc (- x ((λ (x) (* x x)) y))] 11 3)
⇒ (- 11 ((λ (x) (* x x)) 3))
⇒ (- 11 (* 3 3))
⇒ (- 11 9)
;Value: 2
```

Let's examine the function below to try to better understand Scheme's scoping rules.

```
(define scope
  (lambda (x y z)
    (define add-to-y
       (lambda (x) (+ x y)))
    (add-to-y z)))

⇒ (scope 1 2 3)
;Value: 5
```

4. Syntactic Sugar

Notice that we use `lambda` a lot – in fact *every* time we write a function. To save us all from writing (and possibly mispelling) `lamda` so much, there's a short-cut.

The following two expressions are equivalent.

```
(define <proc>                                ⟸⟹      (define (<proc> <arg1> <arg2> ... <argn>)
  (lambda (<arg1> <arg2> ... <argn>)                       <body>)
     <body>))
```

For example, instead of writing: ⟸⟹ we can write:

```
(define average                               ⟸⟹      (define (average x y)
  (lambda (x y)                                            (/ (+ x y) 2))
    (/ (+ x y) 2)))
```

They are the same, so feel free to use either one. If the syntactic sugar is confusing, then stick with the original for now. The important thing to realize is that in both cases, **the lambda expression is evaluated and then bound to the name**, even if `lambda` isn't written anywhere.

5. A Recursive Procedure

Consider the mathematical definition of factorial:

$$n! = 1 \quad \text{for } n = 0$$
$$n! = n(n-1)! \quad \text{for } n > 0$$

Let's write the function `factorial` in Scheme (using the "Syntactic Sugar" as described above):

```
(define (factorial n)
     (if (= n 0)
        1
        (* n (factorial (- n 1))))

)
```