Tutorial 2

SCHEME on CDF

- Invoking: scheme

- Exiting: (exit)

- Loading files: (load “filename”)

- Tracing procedures: (trace proc_name)

- Debugger:
  -- start: (debug)
  -- help: ?
  -- go back: (restart 1) OR ‘ctrl C ctrl C’

Defining Procedures:

(lambda (var1 var2 … varN) exp1 exp2 … expM)

Example:
(lambda (x) (+ x x))
get: Value 1: #[compound-procedure 1]

Evaluating:
((lambda (x) (+ x x)) 3)
get: Value: 6

BUT:
((lambda (x y) (* x y) (+ x y)) 3 5)
get: Value: 8

i.e. ONLY the last expression in lambda is eval'd; can use this for "chatty output" (debugging)

Giving a **name** to a procedure:

```
(define my_proc (lambda (x) (+ x x)))
(my_proc 3)
```
get: Value: 6

Helper procedures:

```
(define my_add
    (lambda (x y)
        (+ x y)))

(define my_avg
    (lambda (x y)
        (/ (my_add x y) 2)))
```

Comments:
Use **;**

Some built-in functions:
(draw an appropriate diagram for each of the following)

1) null?

(null? ())	#t
(null? 1)	#f or ()

2) car

(car '(1 2 3))	1
(car '((1) (2) (3)))	(1)
(car 123)	error

3) cdr

(cdr '(1 2 3))	(2 3)
(cdr '((1) (2) (3)))	((2) (3))
(cdr '(1))	()
(cdr 123)	error

4) cadr, caddr, cadar, etc

(cadr '(1 2 3))	2
(caddr '(1 2 3))	3
(cadar '((1 2) (3))	2

5) cons

(cons 1 ())	(1)	
(cons 1 '(2 3)))	(1 2 3)	
(cons '(1) '(2 3))	((1) 2 3)	
(cons 1 2)	(1 . 2)	(called **pair**)
(cons () ())	(())	

6) quote or '

(quote (1 2 3))	(1 2 3)
'(1 2 3)	(1 2 3)

7) list

(list 1 2 3)	(1 2 3)
(list 1)	(1)

8) append

(append ‘(1) ‘(2))	(1 2)
(append ‘(1 2) ‘(3 4 5)	(1 2 3 4 5)

How Scheme works:

1) procedure call follows ‘(‘
2) evaluate arguments first
3) then evaluate expression

((cadr ‘(+ / -)) 8 2)	error
((cadr (list + / -)) 8 2)	4

why?

‘+	+
+	#[arity-dispatched-procedure 13]

cond

```
(cond       (cond1      expr1)
            (cond 2     expr2)
            …
            (else       exprN) )
```

Scheme on CDF uses lazy-evaluation:

```
(cond       ( (< 1 2)  ‘a)
            ( (= 1 1)  ‘b)
            (else      ‘c) )
```

get: Value: a

Fibonacci numbers:

Fib (0) = 1
Fib (1) = 1
Fib (i) = Fib (i-1) + Fib (i-2)

```
; (fib n) returns the nth Fibonacci number
; Pre: n is a non-negative integer

    (define fib
       (lambda (n)
          (cond ((= n 0) 1)
                ((= n 1) 1)
                (else (+ (fib (- n 1)) (fib (- n 2))))
          )
       )
    )
```