MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.090—Building Programming Experience
IAP 2007

Problem Set 3
Due Tuesday January 16, 1pm

Special Forms

1. **let** – `(let` ((*name1 val1*) (*name2 val2*)) *expr*)
   Let creates a temporary binding between a name and a value that exists only in the body expression. The following two examples are equivalent:

```
(let ((a 5)              ((lambda (a b) (+ a b))
      (b 6))              5 6)
   (+ a b))
```

2. **define** – `(define (`*procname arg1 arg2*`)` *body*`)` Creates a procedure and assigns it to the name *procname*. Syntactic sugar for the following:
   `(define` *procname* `(lambda (`*arg1 arg2*`)` *body*`))`

List Procedures

1. *(list arg1 arg2 ...)* makes a list out of each arugment and returns it.
2. *(cons a b)* Creates a pair, of cons cell, where the car-part is *a* and the cdr-part is *b*.
3. *(car p)* – Takes a pair, and returns the first part.
4. *(cdr p)* – Takes a pair, and returns the second part.
5. *(null? lst)* – Returns #t if the list is the special value for an empty list: `null`

Problems

Section 1: Recursive Processes

1. Write a procedure, num-digits, that takes in a number and returns the number of decimal digits in the number. For example:

```
(num-digits 5)
;Value: 1
(num-digits 21)
;Value: 2
(num-digits 3987423)
;Value: 7
```

You will find `quotient` useful, as it works like `/`, except it throws away anything after the decimal point (ie integer division). For example:

```
(quotient 4 2)
;Value: 2
(quotient 5 2)
;Value: 2
```

First write out a plan (base case and recursive case), then implement the procedure in scheme. Finally, test your procedure to verify that it works.

2. Implement `divisible?`, which returns true if the first argument is divisible by the second. Using remainder is a good idea.

```
(divisible? 6 3)
;Value: #t
(divisible? 37 4)
;Value: #f
```

3. Write a procedure to find the smallest factor of a number. As a suggestion, have the procedure take in both the number n, and a factor f. The procedure should test to see if f is a factor, along with all possible factors greater than f. Some numbers may not have any factors in this range, if so, the procedure should return false. You should use divisible? from the previous problem.

```
(smallest-factor 8 3)
;Value: 4   (4 divides 8 and is between 3 and 8)
(smallest-factor 12 7)
;Value: #f  (12 has no factors larger than 7)
```

4. Building on the previous exercise, write prime?, which returns true if the number is prime. Remember that a prime number has no factors other than 1 and itself.

```
(prime? 4)
;Value: #f
(prime? 17)
;Value: #t
(prime? 569)
;Value: #t
```

Section 2: Syntactic Sugar

1. Desugar the following expressions:

```
(define (foo x)
  (+ x 5))

(let ((x 1))
  x)

(let ((foo (= x 1))
      (bar 7))
  (if foo
      bar
      #f))

(define (weird x y z)     ; this one's odd
  (lambda (foo)
    (+ x y z foo)))
```

2. Evaluate the following expressions (first guess, then check with DrScheme).

```
(define x 5)

(define (y) (+ 7 7))


(let ((x 3))
  (+ x x))


(let ((x (y))
      (y 7))
  (if (> x 3)
      7
      y))

(let ((mit 12))
  (let ((is (+ mit 1)))
    (let ((hard (- is 7)))
      (+ mit is hard))))
```

Section 3: Iterative Procedures

Here is the transformation of fact from a recursive to an iterative process that we did in class:

```
; recursive fact
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))

; iterative fact
(define (fact n)
  (fact-helper n 1))

; helper for iterative version
(define (fact-helper n answer)
  (if (= n 0)
      answer
      (fact-helper (- n 1) (* n answer))))
```

Consider the following recursive definition of quotient:

```
(define (quotient x y)
  (if (< x y)
      0
      (+ 1 (quotient (- x y) y))))
```

Rewrite quotient to give rise to an iterative process by following the pattern we used for fact. Test your resulting procedure to make sure it works like the original.

Section 4: Lists

1. Write expressions whose values print out like the following:

```
(7)

("this" "is" "yummy")

((()))

(("apples" 3) ("oranges" 2))
```

2. `# Here is the length procedure we wrote in class:`

```
Plan:  Base case: empty-list -> length is 0
       Recursive: length whole lst = 1 + length rest of lst

(define (length lst)
  (if (null? lst)
      0
      (+ 1 (length (cdr lst)))))
```

   Write a procedure called sum-list which takes in a list of numbers and outputs their sum.

```
(sum-list (list 1 2 3))
;Value: 6
(sum-list (list 7))
;Value: 7
(sum-list null)
;Value: 0
```

3. Extra Optional Bonus Problem: Write a procedure seven-on-the-end which takes in a list and returns a new list with 7 on the end.

```
(seven-on-the-end null)
;Value: (7)
(seven-on-the-end (list 4))
;Value: (4 7)
(seven-on-the-end (list 4 7 5 3))
;Value: (4 7 5 3 7)
```

   Tackle this problem by figuring out the base case, then the one-off base case (ie where the first recursive call results in the base case).