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

Problem Set 2
Due Friday January 12, 1pm

Scheme Review

1. **Basic Elements**

   (a) *self-evaluating* – expressions whose value is the same as the expression

   (b) *names* – Name is looked up in the symbol table to find the value associated with it.

2. **Combination**
   (*procedure arguments-separated-by-spaces*)
   Value is determined by evaluating the expression for the procedure and applying the resulting value to the values of the arguments.

3. **Special Forms**

   (a) *define* – (`define` *name value*)
   The name is bound to the result of evaluating the value. Return value is *unspecified*

   (b) *if* – (`if` *test consequent alternative*)
   If the value of the test is **not false** (#f), evaluate and return the consequent, otherwise evaluate and return the alternative.

   (c) *lambda* – (`lambda` *parameters body*)
   Create a procedure with the given parameters and body. Parameters is a list of names of variables. Body is one or more scheme expressions. When the procedure is applied, the body expressions are evaluated in order and the value of the last one is returned.

Problems

1. *Evaluation* - For each expression:

   (a) Write the type of the expression

   (b) Write you guess as to the expression's return value. If the expression is erroneous simply indicate "error" for the value. If the expression returns an unspecified value, write whatever you want! If the expression returns a procedure, indicate "procedure" for the value.

   (c) Evaluate the expression, and copy the response from the Interactions Window. Comment out your response to this question.

   ```
   (lambda (x y z) x)
   ```

```
((lambda (happy tiger) (if (= tiger 4) (+ happy tiger) happy))
 4 5)


((lambda (wow this works) (wow this works))
 - 7 5)


((lambda (wow this works) (wow this works))
 7 - 5)


((if (= 4 5)
     (lambda (x) (+ x 3))
     (lambda (x) (+ x 5)))
 7)


(define x 2)
((lambda (x) (+ x x)) 5)


((lambda (yummy) (* yummy yummy)) 5)
yummy
```

2. *Writing Procedures* - For each problem, write the specified procedure while obeying the given constraints on what primitive procedures are available. Additionally, test the procedure with a couple of inputs and include these test cases and their results in your submission to demonstrate that your procedure works.

    (a) Write a procedure `sign` that returns 1 if it's input is positive, -1 if it's input is negative, and 0 if it's input is 0.

```
(define sign
```

    (b) Write a procedure that when given a width, returns the length of the most beautiful rectangle having that width. According to studies, the most beautiful rectangle is one whose ratio of length to width is the golden ratio. The golden ratio can most easily be expressed as $(sqrt(5) + 1)/2$.

```
(beautiful-rectangle 1)
;Value: 1.618033988749895
(beautiful-rectangle 34.5)
```

```
;Value: 55.82217261187137

(define beautiful-rectangle
```

(c) Write a procedure that computes the positive root of the quadratic polynomial using the quadratic formula. The positive root is the larger of the two roots. If the polynomial has complex roots, your procedure should return the string "complex roots".

In the example that follows, the first expression corresponds to solving the equation $x^2 - 2x + 1 = 0$, and the second is $x^2 - 4 = 0$.

```
(positive-root 1 -2 1)
;Value: 1
(positive-root 1 0 -4)
;Value: 2
(positive-root 3 1 3)
;Value: "complex roots"
```

3. *Writing Recursive Procedures* For each problem, first write down your plan (base case, recursive case) in english/math, then write the specified procedure while obeying the given constraints on what primitive procedures are available. The plan should be included with your answer, but commented out using semicolons. Additionally, test the procedure with a couple of inputs and include these test cases and their results in your submission to demonstrate that your procedure works.

(a) Write a procedure `slow-mul` that multiplies two non-negative numbers together using only addition and subtraction (ie not * or /).

```
(slow-mul 3 4)
;Value 12

plan - Base case:
Recursive case: x * y = ((x-1) * y) + y

(define slow-mul
```

(b) Write a procedure `even?` that returns true if it's input is even. You may only use numerical comparisons and subtraction in your solution.

```
(even? 35)
; Value: #f

Plan - base case: if n < 2, n is even if n=0
        recursive case:

  (define even?
```

(c) Write a procedure `slow-remainder` that takes two positive integers a and b, and returns the remainder without using division, multiplication, or the built-in remainder procedure.

```
(slow-remainder 5 2)
;Value: 1

Plan - base case:
        recursive case:
```