Ejercicios de la primera clase (SICP and Bleazley)

Se dan los siguientes expresiones:

```
10

(+ 5 3 4)

(- 9 1)

(/ 6 2)

(+ (* 2 4) (- 4 6))

(define a 3)

(define b (+ a 1))

(+ a b (* a b))

(= a b)

(if (and (> b a) (< b (* a b)))
    b
    a)

(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))

(+ 2 (if (> b a) b a))

(* (cond ((> a b) a)
         ((< a b) b)
         (else -1))
   (+ a 1))
```

1) *Se pide que se de el resultado que imprime el interprete por pantalla*
La lista de resultados es la siguiente: 10 12 8 3 6 19 false 4 16 16

A menudo es necesario calcular expresiones complejas y transcribirlas a Scheme, por ejemplo:
$\frac{5+4+(2-(3-(6+4/5)))}{3*(6-2)*(2-7)}$

2) *Se pide el código en Scheme*
(/ (+ 5 4 (- 2 (- 3 (+ 6 (/ 4 5))))) (* 3 (- 6 2) (- 2 7)))

Aunque el lenguaje Scheme es complicado, en realidad se basa en unas pocas reglas básicas.

3) *Defina una función en Scheme que tome tres números y devuelva la suma de los cuadrados de los dos mayores*

```
(define (cuadrados_mayores x y z)
    (cond ((and (< x y) (< x z)) (+ (* y y) (* z z)))
          ((and (< y x) (< y z)) (+ (* x x) (* z z)))
          ((and (< z x) (< z y)) (+ (* x x) (* y y)))
     )
   )
```

Una observación que no se ha profundizado lo suficiente es que los retornos pueden ser cualquier cosa, números, cadenas de texto e incluso operadores. El siguiente código es totalmente válido:

```
(define (a-plus-abs-b a b)
  ((if (> b 0) + -) a b))
```

4) *Sabiendo que nuestro modelo permite la combinación de funciones, describa que hace el código anterior.*
Esta función tiene 2 funcionamientos distintos dependiendo de por que rama del IF entremos, aunque luego veremos que esos 2 IF se pueden simplificar con una única expresión. Primero, en la condición

del IF se evalua si b es mayor estrictamente a 0, y si eso es así se retorna el valor a + b. Por otro lado, si b es menor o igual a 0, el resultado es a - b. Aquí podemos ver que realmente las dos ramas se comportan igual. Puesto que en la primera rama del IF los numeros son positivos y se suman. En la segunda rama, se resta un número negativo, que es lo mismo que sumarlo en positivo: a - (-b) = a + b. Por lo que la función podría perfectamente simplifiarse con esta expresión : **a + |b|**

El uso de los diferentes órdenes de aplicación y los diferentes resultados es el objeto del próximo ejercicio. Supongamos que tenemos el siguiente código:

```
(define (p) (p))

(define (test x y)
  (if (= x 0)
      0
      y))
```

y evaluamos

```
(test 0 (p))
```

5) *Se pide explicar el comportamiento si se usa el orden aplicativo de evaluación o si usa el orden normal de evaluación.*

Primer caso : orden normal

En este caso, no se evaluan los argumentos hasta que no se utilizan. Por lo que, al escribir la operación (test 0 (p)) no evaluará ni el 0 ni (p). lo que hará será entrar directamente al evaluar el IF, y concluirá que la rama x = 0 tiene como resultado true y que tiene que entrar por esa rama, y entonces en ese momento evaluará el 0 y lo retornará. Por lo que el programa finalizará sin haber evaluado (p), ya que no hizo falta.

Segundo caso : orden aplicativo

En este caso, se evaluan primero los argumentos y luego se evalua el método. Por lo que el orden de evaluación debería ser primero el 0, luego (p) y por último el metodo test. El 0 lo evaluará sin ningun problema, luego, para intentar evaluar (p) el interprete irá a la definición *(define (p) (p))* y (p) será evaluado como (p), que será evaluado como(p), que a su vez también será evaluado como(p), y el interprete entrará en un bucle infinito y nunca podrá poder terminar de evaluar (p), en consecuencia tampoco podrá evaluar el método.