Ejemplos Expresiones Scheme

Definiciones:

```
(define pi 3.141592)

(define perimetro  (lambda (radio) (* 2 pi radio)))
(define superficie (lambda (radio) (* pi (cuadrado radio))))

(define cuadrado   (lambda (x) (* x x)))
```

Evaluaciones:

```
> pi
3.141592

> (perimetro 5)
31.41592

> (superficie 5)
78.5398
```

Definición de Funciones Recursivas

Ejemplo: Función Factorial

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

Evaluaciones:

```
> (factorial 5)
120

> (factorial 40)
815915283247897734345611269596115894272000000000
```

Definición de Funciones

Notación alternativa para definir funciones:

```
(define (perimetro radio) (* 2 pi radio))
(define (superficie radio) (* pi (cuadrado radio)))

(define (cuadrado x) (* x x))

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

Ejemplo: Números Perfectos

Función para determinar si un número es perfecto:

```
(define (perfecto? n)
  (= (suma-divisores n) (doble n)))

(define (doble x) (* 2 x))

(define (suma-divisores n)
  (acumula-divisores-desde n 1))

(define (acumula-divisores-desde x k)
  (cond ((> k x) 0)
        ((divisor? k x)
         (+ k (acumula-divisores-desde x (+ k 1))))
        (else
         (acumula-divisores-desde x (+ k 1)))))

(define (divisor? a b)
  (= (remainder b a) 0))
```

Ejemplo: Números Perfectos

Evaluaciones:

```
> (perfecto? 6)
#t

> (perfecto? 10)
#f

> (perfecto? 28)
#t
```

Expresiones literales

Expresiones precedidas con comilla simple (') son literales, no se evalúan:

```
> pi
3.141592

> 'pi
pi

> (cuadrado 5)
25

> '(cuadrado 5)
(cuadrado 5)
```

Pares ordenados y Listas

Ejemplos de evaluaciones utilizando funciones primitivas:

```
> (cons 'a 'b)
(a . b)

> (car (cons 'a 'b))
a

> (cdr (cons 'a 'b))
b

> (cons 'a (cons 'b (cons 'c '())))
(a b c)

> (list 'a 'b 'c)
(a b c)

> (cadr '(a b c))
b
```

Pares ordenados y Listas

Ejemplos de evaluaciones utilizando funciones primitivas (cont.):

```
> (list 'pi pi)
(pi 3.141592)

> (cadr '(a b c d e))
b

> (cadddr '(a b c d e))
d

> (append '(a b c d e) '(1 2 3))
(a b c d e 1 2 3)

> (reverse (append '(a b c d e) '(1 2 3)))
(3 2 1 e d c b a)
```

Comparación de igualdad

Ejemplos de comparaciones con eq?, eqv? y equal?

```
> (eq? 'a 'a)
#t

> (eq? 'a 'b)
#f

> (eq? '(a b) '(a b))
#f

> (eqv? '(a b) '(a b))
#f

> (equal? '(a b) '(a b))
#t

> (equal? '(a (b c) (d e)) (list 'a '(b c) '(d e)))
#t
```

Comparación de igualdad

Comparaciones entre números: Utilizar la primitiva =

```
> (eq? (+ 1 2) 3)
#t

> (eq? (+ 1.0 2.0) 3.0)
#f

> (eqv? (+ 1.0 2.0) 3.0)
#t

> (eq? (+ 1 2) 3.0)
#f

> (eqv? (+ 1 2) 3.0)
#f

> (= (+ 1 2) 3.0)
#t
```

Árboles Binarios

Definiciones para representar un árbol binario:

- Representación de árbol vacío:

```
(define arbol-vacio '())
```

- Constructores:

```
(define (crear-arbol raiz izq der)
  (list raiz izq der))

(define (crear-hoja raiz)
  (crear-arbol raiz arbol-vacio arbol-vacio))
```

- Selectores:

```
(define (raiz arbol) (car arbol))
(define (izq arbol) (cadr arbol))
(define (der arbol) (caddr arbol))
```

Árboles Binarios

- Funciones de consulta (predicados):

```
(define (vacio? arbol)
  (eq? arbol arbol-vacio))

(define (es-hoja? arbol)
  (and (not (vacio? arbol))
       (vacio? (izq arbol))
       (vacio? (der arbol))))
```

Ejemplo de árbol binario:

```
(define AB
  (crear-arbol
   'a
   (crear-arbol 'b (crear-hoja 'd) (crear-hoja 'e))
   (crear-arbol 'c arbol-vacio (crear-hoja 'f))
   )
 )
```

Árboles Binarios

Resolución de ejercicios usando las funciones definidas:

```
(define (preorden arbol)
  (if (vacio? arbol)
      '()
      (append (list (raiz arbol))
              (preorden (izq arbol))
              (preorden (der arbol)))
   ))

(define (profundidad arbol)
  (if (vacio? arbol)
      0
      (+ (max (profundidad (izq arbol))
              (profundidad (der arbol)))
         1)
   ))
```

Árboles Binarios

Evaluaciones:

```
> (preorden AB)
(a b d e c f)

> (profundidad AB)
3
```

Abstracción de Datos:

- Definir un conjunto básico de operaciones para cada tipo de dato y utilizar sólamente dichas operaciones para manipular los datos.

Ventajas de utilizar este enfoque:

- Nombres de funciones resultan naturales al dominio del problema.
- Genera barreras de abstracción entre los programas que implementan y aquellos que utilizan los tipos de datos.
- Permite cambiar fácilmente la representación utilizada.