

programación, se presenta al computador la función que se desea calcular y este proveerá una respuesta para dicha función. Este proceso se representará mediante la siguiente estructura:

> (*f arg1 arg2 ... argN*)

Resultado

Gran parte del proceso de solucionar un problema de programación consiste en la definición y modelación de las funciones que se desean construir. Una vez construidas se utilizarán sus respuestas para brindar una solución al problema.

Operaciones numéricas elementales

Una de las operaciones más sencillas es sumar números. En general se sabe que:

$$5 + 4 = 9$$

$$5 + 4 + 3 = 12$$

$$5.1 + 4.2 + 3.2 = 12.5$$

En el ambiente de programación que se ha escogido estas operaciones deben representarse por medio de la función suma. A continuación se presentan algunos ejemplos:

> (+ 5 4)

9

> (+ 5 4 3)

12

> (+ 5.1 4.2 3.2)

12.5

Observe la notación que se utiliza. Primero se presenta la función suma y posteriormente los argumentos sobre los cuales se debe operar dicha función. El primer ejemplo posee dos argumentos, los otros dos ejemplos poseen tres argumentos.

Al igual que existe la función suma, existen otras funciones que nos permiten realizar cualquier cálculo numérico. Se describirán algunos ejemplos de las otras funciones numéricas.

Ejemplos de la multiplicación:

```
> (* 2 4)  
8  
> (* 2 4 3)  
24  
> (* 2.1 4.2 3.7)  
32.634
```

Ejemplos de la substracción.

```
> (- 7 4)  
3  
> (- 7 4 1)  
2  
> (- 7.1 4.1 2)  
1
```

Ejemplos de la división.

```
> (/ 10 2)  
5  
> (/ 10 3)  
3.3333...  
> (/ 10 2 2)  
2.5  
> (/ 10.1 2 2.1)  
2.404761904761...
```

Cualquier cálculo que se desee realizar deberá presentarse en el formato expuesto. Por ejemplo:

Para obtener el resultado de una expresión como:

$$(2 * 3 * 4) + (5 * 4) = 44$$

Se debe escribir de la forma:

```
> (+ (* 2 3 4) (* 5 4))  
44
```

Para obtener el resultado de una expresión como:

$$(2 * (5 + 4)) / 3 = 6$$

Se debe escribir de la forma:

```
> (/ (* 2 (+ 5 4)) 3)  
6
```

Operaciones numéricas adicionales

Existen otro conjunto de operaciones numéricas. A continuación se presentan cada una de ellas.

La función (*abs arg1*) se utiliza para encontrar el valor absoluto de un número.

```
> (abs 0)  
0  
> (abs 5)  
5  
> (abs -5)  
5
```

La función (*min arg1 ... argN*) devuelve el mínimo de los argumentos recibidos:

```
> (min 0 3)  
0  
> (min 4 3 1 7)  
1  
> (min 1 2 3 4 3 2 1)  
1
```

La función (*max arg1 ... argN*) devuelve el máximo de un conjunto de argumentos.

```
> (max 1 7)  
7  
> (max 1 7 5 )  
7  
> (max 1 2 3 4 5 3 2)  
5
```

La función (*truncate argI*) trunca el valor de un número real hacia el entero inmediatamente inferior.

```
> (truncate 10.33)  
10  
> (truncate 10.50)  
10  
> (truncate 10.78)  
10
```

La función (*round argI*) redondea el valor de un número a su valor entero más cercano:

```
> (round 10.33)  
10  
> (round 10.50)  
11  
> (round 10.78)  
11
```

La función (*quotient arg1 arg2*) devuelve la división entera de sus argumentos. Los argumentos deben ser números enteros.

```
> (quotient 10 2)  
5  
> (quotient 10 3)  
3
```



```
> (quotient 10 4)
```

```
2
```

```
> (quotient 10 5)
```

```
2
```

La función (*remainder arg1 arg2*) devuelve el residuo que se obtiene al dividir arg1 entre el arg2. Su resultado tiene el mismo signo que arg1. Sus argumentos deben ser números enteros.

```
> (remainder 10 2)
```

```
0
```

```
> (remainder 10 3)
```

```
1
```

```
> (remainder 10 4)
```

```
2
```

```
> (remainder 10 5)
```

```
0
```

```
> (remainder 10 -4)
```

```
2
```

La función (*expt arg1 arg2*) devuelve el valor de arg1 elevado según arg2.

```
> (expt 2 1)
```

```
2
```

```
> (expt 2 3)
```

```
8
```

```
> (expt 2.11 2)
```

```
4.4521
```

```
> (expt 2.11 3.12)
```

```
10.274513344166518
```

La función (*sqrt arg1*) devuelve la raíz cuadrada de arg1.

```
> (sqrt 1)
```

```
1
```

```
> (sqrt 2)
```

```
1.4142135623730951
```

```
> (sqrt 4)
```

```
2
```

```
> (sqrt 5)
```

```
2.23606797749979
```

Predicados numéricos

Los predicados son funciones que devuelven únicamente el valor de verdadero o falso. El valor de verdadero se representa mediante el símbolo de #t y el valor de falso se representa mediante el símbolo de #f.

A continuación se presentan los principales predicados del lenguaje.

Existe una función para corroborar si dos números o más son iguales. Esta función se denomina ($= arg1 arg2 \dots argN$) y necesita por lo menos de dos argumentos. Esta función devuelve el valor de #t si todos sus argumentos son iguales.

```
> (= 7 7)
```

```
#t
```

```
> (= 0 (- 7 7))
```

```
#t
```

```
> (= 0 (- 7 6))
```

```
#f
```

```
> (= 0 (- 7 7) (- 6 (+ 2 4)))
```

```
#t
```

Existe una función similar denominada ($equal? arg1 arg2$), en el caso que sus argumentos sean números funciona igual que ($= arg1 arg2$), con la diferencia que recibe únicamente dos argumentos.

```
> (equal? 7 7)
```

```
#t
```

```
> (equal? 0 (- 7 7))
```

```
#t
```

```
> (equal? 0 (- 7 6))
#f
```

Para corroborar la relación de orden que existe entre dos o más elementos se utilizan las funciones:

Para menor estricto ($< arg1\ arg2\ ... argN$)

Para menor igual ($<= arg1\ arg2\ ... argN$)

Para mayor estricto ($> arg1\ arg2\ ... argN$)

Para mayor igual ($>= arg1\ arg2\ ... argN$)

Todos estos predicados requieren por lo menos de dos argumentos.

```
> (< 2 7)
```

```
#t
```

```
> (< 7 10)
```

```
#t
```

```
> (< 10 7)
```

```
#f
```

```
> (<= 2 7)
```

```
#t
```

```
> (<= 7 7)
```

```
#t
```

```
> (<= 10 7)
```

```
#f
```

```
> (<= 2 4 6 8 10)
```

```
#t
```

La función ($zero?\ arg1$) recibe un argumento y devuelve el valor de #t si ese argumento es igual a zero y #f en cualquier otro caso.

```
> (zero? 0)
#t
```

```
> (zero? 7)
#f
```

```
> (zero? (- 7 7))
```

```
#t
```

```
> (zero? (- 7 6))
```

```
#f
```

La función (*positive?* *arg1*) recibe un argumento y devuelve el valor de #t si ese argumento es mayor que zero y #f en cualquier otro caso.

```
> (positive? -2)
```

```
#f
```

```
> (positive? 0)
```

```
#f
```

```
> (positive? 5)
```

```
#t
```

La función (*negative?* *arg1*) recibe un argumento y devuelve el valor de #t si ese argumento es menor que zero y #f en cualquier otro caso.

```
> (negative? -2)
```

```
#t
```

```
> (negative? 0)
```

```
#f
```

```
> (negative? 5)
```

```
#f
```

La función (*even?* *arg1*) devuelve el valor de verdadero #t si su argumento es un número par y devuelve el valor de falso #f si su argumento es un número impar.

```
> (even? 2)
```

```
#t
```

```
> (even? 3)
```

```
#f
```

```
> (even? 4)
```

```
#t
```

La función (*odd?* *arg1*) devuelve el valor de falso #f si su argumento es par y devuelve el valor de verdadero #t si su argumento es impar.

```
> (odd? 2)
#f
```

```
> (odd? 3)
#t
```

```
> (odd? 4)
#f
```

La función (*number?* *arg1*) recibe un argumento y devuelve el valor de #t si ese argumento es un número y #f en cualquier otro caso.

```
> (number? 5)
#t
```

```
> (number? 5.11)
#t
```

La función (*integer?* *arg1*) devuelve el valor de #t si se encuentra con un número entero.

```
> (integer? 4)
#t
```

```
> (integer? 4.0)
#t
```

```
> (integer? 4.77)
#f
```

La función (*real?* *arg1*) devuelve el valor de #t si se encuentra con un número real.

```
> (real? 5)
#t
```

```
> (real? 5.0)
#t
```

Predicados lógicos

Los predicados lógicos son funciones que implementan las principales conectivas de la lógica simbólica. En general reciben como entradas valores de #f y #t y producen uno de esos valores como su salida.

El predicado (*and arg1 arg2 ... argN*) devuelve el valor de #t únicamente si todos sus argumentos son verdaderos.

```
> (and)
#t

> (and #t #t)
#t

> (and #t #f)
#f

> (and #f #t)
#f

> (and #f #f)
#f

> (and (= 7 7) (zero? (- 7 7)) #t)
#t
```

El predicado (*or arg1 arg2 ... argN*) devuelve el valor de #t si alguno de sus argumentos es verdadero.

```
> (or)
#f

> (or #t #t)
#t

> (or #t #f)
#t

> (or #f #t)
#t

> (or #f #f)
#f

> (or (= 8 7) (zero? (- 7 7)) #f)
#t
```

El predicado (*not arg1*) devuelve el valor de #t si su argumento es falso y devuelve el valor de #f si su argumento es verdadero.

```
> (not #t)
#f

> (not #f)
#t

> (not (zero? 1))
#t
```

Las funciones COND y DEFINE

La función *cond* es un tipo especial de función que permite seleccionar una opción de múltiples alternativas. La sintaxis de esta función es:

```
> (cond (condición-1?
          función-1)
        (condición-2?
          función-2)
        ...
        (condición-N?
          función-N) )
```

Esta función evalúa cada una de las condiciones y cuando encuentra la primera condición verdadera realiza la función asociada.

```
> (cond ( (=7 (+ 34) )
          #t)
        ( (= 7 5))
          #f) )
```

#t

```
> (cond ( (=7 (+ 34) )
          #t)
        ( else
          #f) )
```

#t

La función *define* puede utilizarse de varias formas, una de ellas sirve para crear variables. La sintaxis para crear variables es:

```
> (define <nombre-variable>  
      <valor-variable> )
```

La función *define* no devuelve un valor específico, en distintos ambientes de programación se producen diferentes valores. En este caso no se producirá nada como resultado.

A continuación se presentan algunos ejemplos para la creación de variables.

```
> (define a 5)
```

```
> a
```

```
5
```

```
> (define b 10)
```

```
> b
```

```
10
```

```
> (define a 7)
```

```
> a
```

```
7
```

```
> b
```

```
10
```

Otra forma de utilizar la función de *define* es para crear funciones. Para crear funciones existen dos estilos, uno mediante el uso de la palabra *lambda*.

```
> (define <nombre-de-la-función>  
      (lambda (arg1 ... argN)  
            <cuerpo-de-la-función>))
```

Por ejemplo, para crear la función sucesor, la cual recibe un número entero y devuelve sucesor se puede construir de la siguiente forma:

```
> (define suc  
      (lambda (num) (+ num 1)))  
  
> (suc 0)  
1
```

```
> (suc 2)
```

```
3
```

```
> (suc 3)
```

```
4
```

La forma abreviada para crear funciones tiene la sintaxis:

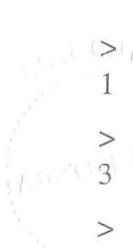
```
> (define (<nombre-de-la-función> arg0 Arg1 ... argN)
```

```
    <cuerpo-de-la-función>)
```

De esta forma, la función anterior se pudo haber construido de la siguiente manera:

```
> (define (suc num)
```

```
    (+ 1 num))
```



```
(> (suc 0)
```

```
1
```

```
> (suc 2)
```

```
3
```

```
> (suc 3)
```

```
4
```

Se suele utilizar *define* en conjunción con la función *cond* para crear nuevas funciones. A continuación se construye la función *abs*. Observe como en la construcción que se presenta se utiliza como última condición el enunciado *else*, que se evalúa siempre como verdadero.

```
> (define (abs num)
  (cond ((> num 0)
         num)
        ((< num 0)
         (* -1 num))
        (else
         0)))
```

Una forma más corta de escribir la función anterior podría ser:

```
> (define (abs num)
  (cond ((>= num 0)
```

```

        num)
( else
  (* -1 num))))

```

La función LET

Es usual que las fórmulas que se desean programar tengan una estructura muy compleja por lo que resulta útil emplear variables intermedias para el cálculo. Para ello se utiliza la función *let*, la cual tiene la siguiente sintaxis:

```

> (let ( <declaración-de-variables>
         )
      <cuerpo-de-la-función>)

```

De forma más detallada, se puede expresar como:

```

> (let ( (variable-1 función-1)
          (variable-2 función-2)
          (variable-3 función-3)
          ...
          (variable-n función-n)
          )
      <cuerpo-de-la-función> )

```

Suponga que se desea programar la siguiente función:

$$f(x,y) = y(1 - y) + (1 - y)(1 + y) + 2x(1 + y)(1 + y)$$

Se podría calcular de la siguiente manera:

```

> (define(f x y
           (+ (* y
                  (- 1 y))
              (* (- 1 y)
                  (+ 1 y)))
              (* 2 x
                  (+ y 1)
                  (+ y 1)))))


```

Sin embargo, resulta mucho más sencillo y eficiente programar dicha función de la siguiente forma:

```
> (define (f x y)
  (let ( (a (- 1 y))
        (b (+ 1 y))
        )
    (+ (* y a)
       (* a b)
       (* 2 x b b))))
```

Existe además otra función denominada *let** que funciona de manera similar a *let*. Su diferencia radica en el hecho que *let* asocia las variables de manera paralela (todas al mismo tiempo) mientras que *let** asocia las variables de manera secuencial (en el orden descrito). A continuación se muestra una diferencia entre estas dos funciones.

```
> (let ( (a (+ 2 4))
         (b (+ 2 a))
         )
      b)
```

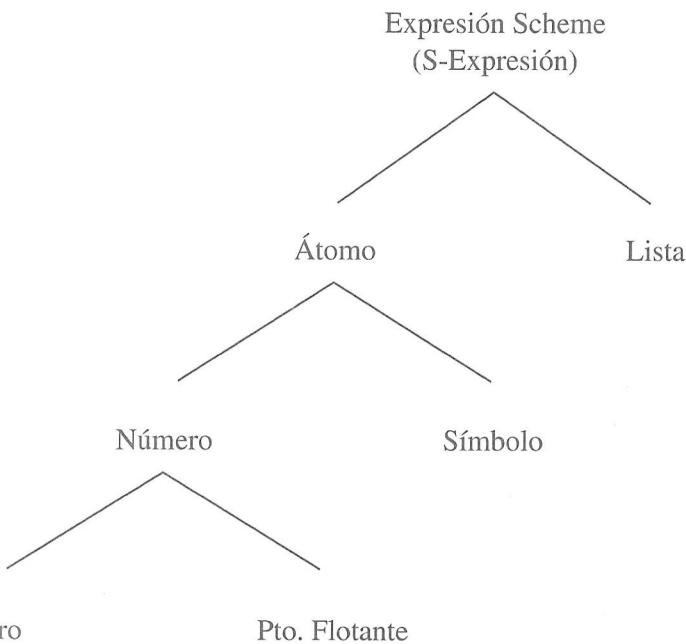
error --> a no está definido

```
> (let* ( (a (+ 2 4))
           (b (+ 2 a))
           )
      b)
```

8

Relaciones entre las expresiones

Se han presentado números, símbolos, listas y funciones propias del ambiente de programación. Se ha demostrado que estos elementos se pueden utilizar para construir nuevos elementos. A continuación se presenta un gráfico que muestra las relaciones que existen entre los distintos tipos de objetos presentados.



La función QUOTE

La función (*quote arg1*) puede escribirse de esta forma, existe una forma abreviada que se escribe como ‘arg1’. Posee el siguiente comportamiento.

```
> (quote 7)
7
> (quote (+ 4 5))
(+ 4 5)
> (quote A)
A
> (quote (zero? 0))
(zero? 0)
> '7
7
> '(+ 4 5)
```

```
(+ 4 5)
```

```
> 'A
```

```
A
```

```
> '(zero? 0)
```

```
(zero? 0)
```



La función APPLY

La función (*apply fun '(arg1 ... argN)*) es utilizada por el ambiente de interpretación. Recibe dos argumentos, el primero es una función y el segundo son los argumentos de la función. Realiza el siguiente proceso:

```
> (apply fun '(arg1 ... argN))
  (fun arg1 ... argN)
```

A continuación se presentan algunos ejemplos del uso del intérprete de Scheme y luego el uso de *apply*.

```
> (+ 1 2 3 4)
  10
```

```
> (apply + '(1 2 3 4))
  10
```

```
> (abs 5)
  5
```

```
> (apply abs '(5))
  5
```

```
> (apply abs '(-5))
  5
```

La función EVAL

Cuando se le presenta una nueva expresión a Scheme, este automáticamente le solicita a la función *eval* para que evalúe dicha expresión. La función (*eval expresión*) es una función usada por el ambiente de interpretación del lenguaje. A continuación se presentan algunos ejemplos:

```
> (eval (quote 7))
```

```

> (eval '(+ 1 2 3))
6
> (eval '(define a 3))
a
> (eval '(define b 4))
b
> (eval '(+ a b))
7

```

Evaluación de las funciones

Hasta ahora se han utilizado números, símbolos, listas y funciones para ser evaluadas por el lenguaje. Cada vez que se presenta uno de estos elementos se produce una respuesta. En el caso de un número la respuesta es el mismo número. En el caso de un símbolo la respuesta es el valor del símbolo. En el caso de una función se produce un proceso de evaluación de los argumentos y posteriormente la aplicación de la función a cada uno de sus argumentos. En el caso de las funciones se han representado por medio de la expresión:

$\star (f \ arg1 \dots \ argN)$

Respuesta

Se presenta ahora el algoritmo de evaluación que se utiliza para producir las respuestas. Para la escritura de dicho algoritmo se supondrá que la expresión $S = (f \ arg1 \dots \ argN)$

Algoritmo de Evaluación:

Si S es un número, entonces devuelva el número.

Si S es un símbolo, entonces devuelva el valor de S .

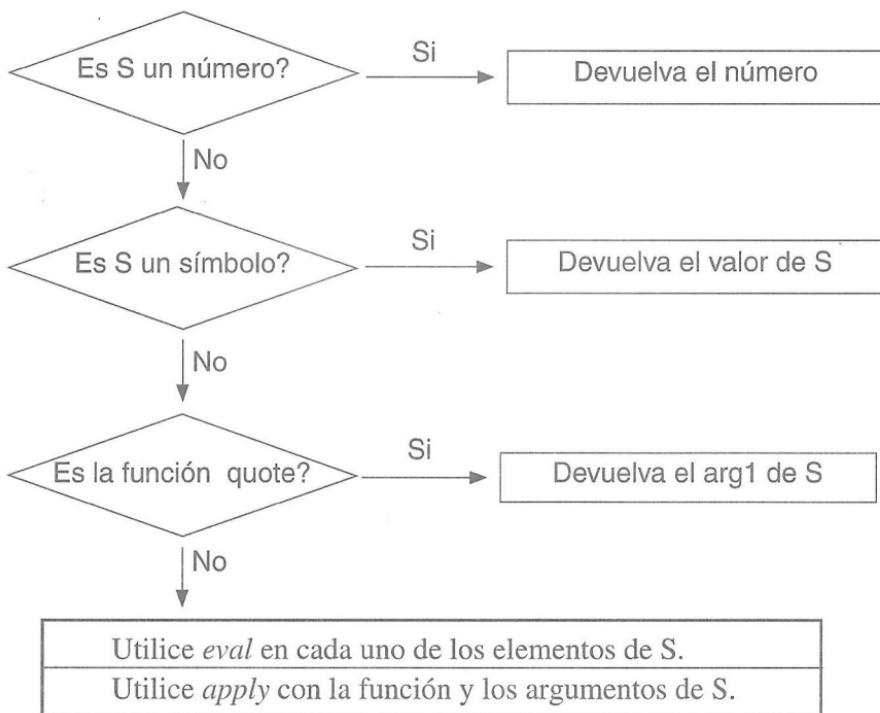
Si es la función quote, devuelva el $arg1$ de S .

En otro caso, Utilice *eval* en cada uno de los argumentos de S

Utilice *apply* con f y el resultado de los argumentos.

El Algoritmo anterior se puede representar gráficamente como:

$S = (f \ arg1 \dots \ argN)$.



Resumen

- Se pueden realizar operaciones de *sumar*, *restar*, *multiplicar* y *dividir* pero se debe utilizar las formas de Scheme.
- Las funciones *abs*, *min*, *max*, *truncate*, *round*, *quotient*, *remainder*, *expt* y *sqrt* son operaciones numéricas auxiliares.
- Las funciones *zero?*, *even?*, *odd?*, *equal?*, *=*, *>*, *>=*, *<*, *<=*, son predicados numéricos que devuelven valores de #t y #f.
- Las funciones *and*, *or*, *not*, son predicados lógicos que reciben y producen valores de #t y #f. Se comportan de acuerdo a las reglas de la lógica simbólica.
- La función *define* es una función muy especial que sirve para crear otras funciones. Se puede utilizar la forma larga o la forma abreviada.
- La función *cond* se utiliza como un método de decisión y selección.
- Las funciones *let* y *let** permiten ligar valores a variables.

- Las funciones *quote*, *apply* y *eval* son funciones especiales que se utilizan por el intérprete.
- El intérprete evalúa números, símbolos y funciones. Utiliza además un algoritmo de evaluación que evalúa primero la función y los parámetros y luego aplica la función a los mismos.

Ejercicios

Ejercicio N° 1

Indique cuál es el resultado si se aplican las siguientes funciones al ambiente de programación.

```
> (/ 1 3)
> (- 10 (- 8 (- 6 4)))
> (/ 40 (* 6 15))
> (+ (* 0.1 20) (/ 4 3))
> (+ (/ 1 3) (/ 2 3))
```

Ejercicio N° 2

Escriba las expresiones en Scheme que realizan los cálculos indicados.

```
(4 * 7) - (20 + 2)
(3 * (4 + -5 + -2))
(2.5 / (5 * (1 / 10)))
(1 / 3) + (4 / 5) + (65 * (18 / 2)) + 21
(5 * 42 * (23 / 2) * (12 + 2)) - (2 * (18 / 3))
```

Ejercicio N° 3

Si a, b y c son tres números cualesquiera, convierta las siguientes expresiones en Scheme a la notación matemática usual. Por ejemplo: $(+ a (* b c)) = a + (b * c)$.

```
> (+ a (- (+ b c) a))
> (+ (* a b) (* b c))
> (/ (- a b) (- a c))
> (* (+ a b c) (- a b c) (/ a b c))
> (/ (* a b) (* b c) (* a c))
```

Ejercicio N° 4

Realice los siguientes cálculos e indique el resultado.

```
> (min 0 (abs (- 8 7)) (abs (- 7 8)))
> (min (max 0 1 2 1) (max 7 5 4 2))
> (truncate (round 7.33))
> (round (abs (* 2.11 3.18)))
> (abs (- (quotient 19 3) (remainder 19 3)))
> (* 3 (remainder 11 -2))
> (* 3 (modulo 11 -2))
```

Ejercicio N° 5

Realice los siguientes cálculos e indique el resultado.

```
> (zero? (- 7 7) (* 0 2))
> (zero? (- 7 6.99))
> (= (* 2 3 5) (* 15 2))
> (= (/ 1 3) 1.333)
> (= (/ 1 3) (- 1 (/ 2 3)))
> (equal? #t #t)
> (equal? (> 8 7) (>= 6 6))
> (and (<= 2 3) (<= 10 12) (<= 20 21))
> (or (> 2 3) (< 10 12) (> 20 21))
```

Ejercicio N° 6

A continuación se presenta un conjunto de instrucciones para el evaluador de Scheme. Si estas instrucciones se introducen en el mismo orden que se presentan, indique cuáles serían los resultados que se producirían al evaluarlos.

```
> (define a 5)
> (quote a)
> 'a
> a
> (+ 5 a)
> (define a 7)
> (+ 5 a)
> (cond ((= a 0) 0) ((> a 0) -1) ((> a 0) 1))
> (define (test n)
  (cond ( (= n 0) 0)
```

```

( (< n 0) -1)
( (> n 0) 1)))
> (test -3)
> (test 0)
> (test 3)
> (test a)
> (define (test a)
  (cond ( (= a 0) 0)
        ( (< a 0) -1)
        ( (> a 0) 1)))
> a
> (test a)

```

Ejercicio N° 7

A continuación se presenta un conjunto de instrucciones para el evaluador de Scheme. Si estas instrucciones se introducen en el mismo orden que se presentan, indique cuáles serían los resultados que se producirían al evaluarlos.

```

> (let ( (a (+ 2 4))
      (b (+ 7 1))
      (c (+ 7 7))
      )
  (+ a b c))

> (let* ( (a (+ 2 4))
      (b (+ 7 1))
      (c (+ 7 7))
      )
  (+ a b c))

> (let ( (a (+ 2 4))
      (b (+ 7 a))
      (c (+ 7 a))
      )
  (+ a b c))

> (let* ( (a (+ 2 4))
      (b (+ 7 a))
      (c (+ 7 a))
      )
  (+ a b c))

```

Ejercicio N° 8

Indique como funciona el algoritmo que evalúa las funciones de Scheme. En especial debe describir cuando se utiliza *quote*, *eval* y *apply*.

Ejercicio N° 9

A continuación se presenta un conjunto de instrucciones para el evaluador de Scheme. Si estas instrucciones se introducen en el mismo orden que se presentan, indique cuáles serían los resultados que se producirían al evaluarlos. Algunas de ellas pueden producir errores.

```
> ((lambda (x y) (+ x y)) 2 3)
> ((lambda (x y) (+ x y)) 4 5)
> (define suma (lambda (x y) (+ x y)))
> (suma 2 3)
> (suma 4 5)
> (define (suma x y) (+ x y))
> (suma 2 3)
> (suma 4 5)
> (apply (lambda (x y) (+ x y)) '(2 3))
> (apply + '(1))
> (apply + '(1 2 3))
> (apply abs '(1))
> (apply abs '(-1 1))
```

Ejercicio N° 10

En su ambiente de programación en Scheme, identifique las dos áreas principales del ambiente. En general se tiene el área de escritura de programas y el área de ejecución.

Escriba las siguientes funciones en el área de escritura de programas. Las líneas que inician con el símbolo “;” se llaman comentarios. Son líneas que NO son evaluadas por el ambiente, sirve únicamente para poner información útil para el programador. Luego ejecútelas en el área de ejecución de programas.

```
; Función para obtener
; el sucesor de un número
;;
```

```
(define (suc num)
  (+ 1 num))
;;
;; Función que calcula
;; el valor absoluto
;;
(define (abs num)
  (cond ((>= num 0)
         num)
        (else
         (* -1 num))))
```