Título

Jorge Ivory Castaño Vergara

25 de agosto de 2003 , versión 1.3

Índice general

Capítulo 1

Estructura de los programas

La forma más fácil de aprender a programar es pensar que estamos aprendiendo otro idioma, un idioma que el computador nos puede entender. De esta forma, el trabajo del programador consiste en darle intrucciones al computador en un lenguaje que él comprende para asi realizar las tareas que le asignan.

En programación trabajamos con funciones o métodos; estas tienen un nombre y un conjunto de instrucciones, de esta manera se nos hace más fácil darle un orden a las instrucciones para asi interactuar con el computador.

Toda la información que usamos debe estar almacenada en el computador para que este pueda realizar las tareas que le asignamos y darnos una respuesta, es por eso que el programador trabaja con dos tipos de objetos, los procedimientos y los datos.

Los PROCEDIMIENTOS los podemos ver como algoritmos, o aquellos pasos que debemos seguir para la ejecución de un proceso.

Los DATOS son las representaciones de las cosas que existen en el mundo real, y sirven para describir las situaciones que se nos presentan en los problemas que queremos modelar en el computador
Para poder representar diferentes situaciones usamos datos de varios tipos: enteros, cadenas de caracteres, simbolos, etc

Algunos ejemplos de tipos de datos son:

Entero(integer): Son los números positivos y negativos, tales como 1, 2, 3, -54, 873, etc.

Numero(number): Son todos los números, incluidos aquellos que tienen punto decimal o son fraccionarios. Ejemplo de números son: 3, -9/8, 5.25.

Cadena de caracteres(String): Es una cadena que puede contener números, espacios, letras y caracteres especiales. Esta cadena debe estar entre co-

millas dobles. Por ejemplo la frase "Favor digite el precio, sin el símbolo $ ", corresponde a un String.

Simbolo(symbol): Un simbolo es Scheme es una palabra, sin espacios. Para diferenciarla del String que puede o no contener espacios, se utiliza una comilla simple al inicio. Por ejemplo 'color 'Jaime, etc son simbolos.

Otros ejemplos de tipos de datos se verán más adelante, en la sección (seccion de Wilches y Tito).

Todo lenguaje de esta compuesto por un alfabeto, un vocabulario y una gramática.

El alfabeto es el conjunto de caracteres aceptados por el lenguaje. En Scheme (y en la mayoría de los lenguajes de progrmamación) estos son:

- Caracteres Alfabéticos: Son las letras, mayúscula y minúsculas.
- Caracteres Numéricos: Números arábigos del 0 al 9.
- Caracteres Especiales: Son signos de puntuación tales como [.], [,], [;], [\], [/], [:], [#], [(], [)], y otros.

El vocabulario es el conjunto de palabras que tienen un significado propio en el lenguaje, y por lo tanto estan reservadas. Por ejemplo, algunas palabras resevadas para Scheme son: define, let, local, begin, end, if, then, else, integer, String, number, entre otras. Cada lenguaje tiene sus propias palabras reservadas

Gramática es la forma en que se deben construir las frases, oraciones, etc para que tengan un significado o expresen una idea. En los lenguajes de programacion son las reglas para escribir un procedimiento (programa). Por ejemplo para que el computador nos entienda debemos obedecer a la sintaxis propia de cada lenguaje (forma de escribir) en Scheme se tiene como modelo el procedimiento encerrado en parentesis, donde el operador es el primer elemento separado de los operandos por un espacio en blanco, como por ejemplo: (+ 2 1), (+ (- 3 2) 9). Este modelo donde primero se encuentra el operador y luego los operandos se conoce como notación prefijo.

(operador operando1 operando2... operandoN)

Debemos tener en cuenta que la cantidad de operandos puede ser 0, 1, 2 o más dependiendo del operador utilizado.
Ejemplo:
Para escribir la instrucción: muliplique 9 por 5:
==>(* 9 5)

Se admiten como operandos cualquier procedimiento valido en Scheme, siempre y cuando tenga sentido con el operador utilizado. Scheme trata los llamados a

las funciones como el resultado que ellas producen, es decir, primero las evalúa y luego trabaja con el resultado de haber evaluado esa expresión. En otras palabras, los procedimientos son ciudadanos de primer orden.

1.1. Un programa en Scheme

Para empezar a trabajar en Scheme, en la ventana inferior podemos ver el prompt o linea de comandos, donde aparecera algo asi como:
==>
Si escribimos una expresion en el prompt, el nos respondera devolviendo el resultado de evaluar tal expresión. Por ejemplo podemos escribir un número
==>19
el intérprete responderá imprimiendo en la pantalla
19
Un procedimiento bastante simple sería por ejemplo, realizar una suma:
==>(+ 120 35)
155
Para escribir un programa es Scheme, tenemos que tener en cuenta que debemos respetar la gramática, es decir, escribir primero la operacion a realizar y luego los operandos con los que vamos a trabajar.
Asi como en algebra, podemos unir expresiones Por ejemplo la operación:

(9+7)/(4*2)
se escribiría en Scheme de la siguiente forma:
(/ (+ 9 7) (4 2))*
y el resultado de evaluar esta expresión sería 2.
(/ (+ 9 7) (4 2))*
(/ 16 8)
2

Scheme no solo permite realizar cálculos algebraicos básicos, sino que también permite el uso de muchas otras funciones, estas son solo algunas de ellas:

*(expt A B)*nos da el resultado de A a la B-ésima potencia;

*(remainder A B)*devuelve el residuo de la operación A dividido B;

*(log A)*devuelve el logaritmo natural de A;

*(sin A)*nos da el seno de A radianes.

Los programas en Scheme son básicamente funciones; a las cuales les asignamos un nombre, tienen parámetros o datos de entrada, una fórmula (operación o algoritmo que debemos aplicarle a estos datos) y un dato de salida. En Scheme la forma de construir un programa es la siguiente:
Se escribe:
(define (nombre-de-la-funcion parametro1 parametro2 parametroN....)

(cuerpo de la funcion))

El número de parámetros no está limitado y deben estar separados por espacios. Hay que tener en cuenta que al escribir los nombres debemos usar las reglas de los identificadores que se explican en la siguiente sección (Sección 1.2.1, pág 9.) y debe ser bastante claro tanto para nosotros como para cualquier otra persona que vaya a leer el programa a qué se refiere cada valor o cada función.

Un buen programador comenta sus programas, esto consiste en agregar comentarios o notas personales que expliquen que hace cada porción del código que se ha escrito, de esta manera se hace más fácil a las otras personas entender lo que hicimos y a nosotros mismos cuando posteriormente trabajemos haciendo mejoras o correcciones en nuestros programas.
Para introducir un comentario en Scheme escribimos punto y coma [;], esto le indica al computador que es una nota y no una instrucción que deba cumplir; así no interpreta el código que está en esa linea.

1.1.1. Ejemplos

Escribir en Scheme, la forma correcta de solucionar los siguientes problemas:
Ejemplo 1:
Escribir una función que sume 2 números

```
(define (sumar a b)
(+ a b ))
```

Para llamar la función, debemos escribir:
(nombre-de-la-funcion primer-parámetro segundo-parámetro ...)
Ejemplo:

```
==>(sumar 3 5)
8
==>
```

Ejemplo 2:
Escribir una función que encuentre la n-ésima potencia de un número

```
(define (potencia numeroBase nesimaPotencia)
(expt numeroBase nesimaPotencia))
```

Para llamar la función, debemos escribir:

```
==>(potencia 2 4)
16
==>
```

Ejemplo 3:

Hallar el valor en moneda local de un monto de moneda extranjera teniendo en cuenta la tasa de cambio y el impuesto a la compra-venta de divisas.

Lo primero que debemos hacer tener en cuenta todos los datos que necesitamos, puesto que son conocidos podemos continuar con el problema
Primero debemos saber el valor de la moneda que queremos cambiar en moneda local y la cantidad .
Por ejemplo: si tenemos 2 dólares y cada uno vale 2830 pesos debemos realizar una multiplicación, 2 x 2830.
Pero debemos tener en cuenta el impuesto a pagar, es decir, un porcentaje sobre el dinero que se va a pagar como impuesto.
Segun este razonamiento debemos:
1. Multiplicar la cantidad por el valor de la moneda
2. Con base en esto, debemos hallar la cantidad a pagar por el impuesto, es decir, multiplicar el resultado por el porcentaje o valor del impuesto(si por ejemplo el impuesto es del 3 % debemos multiplicar por 3 y dividir por 100).
3. Restar de la cantidad a pagar el total deducido del impuesto.

Nuestra funcion en Scheme quedaría así:

```
(define (cambiarAPesos cantidadDeLaMoneda valorDeLaMoneda impuesto)
  (-
    (* cantidadDeLaMoneda valorDeLaMoneda)
     (/ (* (* cantidadDeLaMoneda valorDeLaMoneda) impuesto)100)))
```

Si por ejemplo queremos cambiar 10 libras cuyo valor es 3500 pesos colombianos por libra, y debemos pagar un impuesto del 10 % debemos escribir lo siguiente en la ventana de interacción.
==>(cambiaraPesos 10 3500 10)
31500

1.2. Identificadores, constantes, variables

1.2.1. Identificadores

Los identificadores son usados para nombrar los objetos de un programa (constantes, variables, tipos de datos, procedimientos, etc). y tienen las siguientes reglas:
Deben comenzar con una letra o el símbolo "_ ", y no pueden contener espacios en blanco

Por ejemplo, si deseamos almacenar en el computador la edad de un estudiante, este seria un dato de tipo entero (integer) y conviene darle un nombre que os indique a qué se refiere, puede ser: edad_Estudiante, edadEstudiante o simplemente edad.

NOTA: Por cuestiones de estilo, para los identificadores se utiliza escribir todo en minúsculas y en mayúsculas solo la primera letra de la palabra que sigue (por la imposibilidad de utilizar espacios), asi: numero de clientes por hora se podría

escribir NumeroClientesPorHora, o de una forma más corta NumClienteHora.

1.2.2. Variables

Las variables tienen un identificador o nombre y un valor asociado, este valor se puede modificar durante la ejecución del programa. Declarar una variable nos permite usar como referencia el nombre de la misma en lugar de su valor.
Para declarar una variable en Scheme usamos:

(define nombre-de-la-variable valor)

Ejemplo:
(define x 10)
;esto le da el valor de 10 a la variable x
==>(+ x 4)
14

Si en algun momento queremos asignarle un valor diferente a una variable que ya ha sido declarada usamos el operador set!:

(set! x 7)
;le asignamos un nuevo valor a x
==>(+ x 4)
11

1.2.3. Constantes

Las constantes son valores que no se cambian durante la ejecución del programa. Por ejemplo, si vamos a trabajar con circulos o funciones trigonométricas nos sería útil asignarle un valor a **pi** para usar esta contante en vez de tener que escribir su valor cada vez que vayamos a utilizarla.
Esto además nos da una ventaja, si queremos trabajar con más precisión solo debemos cambiar en el código la linea donde le asignamos el valor a **pi** (por ejemplo poner 3.1416 en lugar de 3.14), si no trabajaramos definiendo esta costante tendríamos que modificar el valor cada vez que éste aparezca en el código.

Ejemplo:

(define pi 3.14)
;esto le da el valor de 3.14 a la variable pi

(define (perimetroCirculo radio)
(*(* 2 pi) radio))

(define (areaCirculo radio)
(* pi(* radio radio)))

Debido a que mientras usamos el programa no vamos cambiar el valor de **pi**, esta se conoce como una constante.

1.3. Construyendo un programa simple

Ya habíamos visto que un programa simple es como una función, a la cual se le define un parámetro de entrada y después de realizar un procedimiento, debe arrojar un resultado.

Por ejemplo, podemos definir la función f así:

f(x)= x+3

En este caso, f es el nombre de la función, x es el parámetro o dato de entrada y la salida o valor de retorno corresponde al resultado de evaluar la expresión.

f(2)=5

Tomando 2 como valor de entrada el resultado nos da 5, que es el valor de la salida o retorno

Debemos hacer también una diferencia entre parámetros formales y parámetros reales. Parámetro formal se refiere al nombre que se le da al dato al definir la función (en este ejemplo, **x** es el parámetro formal). Y el parámetro real se refiere al dato que se usa para evaluar la función (en este caso, el **2**).

Ejemplo: Si se nos pide calcular cuanto dinero se debe pagar a cada uno de los empleados de un almacén, si el salario es de 120 por cada hora trabajada.

Nombre	Horas Trabajadas
Juan	80
Luis	72
Andrea	81
Jimena	65

Lo primero que debemos hacer es pensar que datos necesitamos y si los tenemos todos, ya que estas van a ser los parámetros de la función. Sabemos cuánto es el salario base y sabemos también cuántas horas trabajó el empleado.
Además sabemos que el salarioBase es el mismo para todos los empleados y que no va a cambiar, por lo tanto lo definimos y usamos como una constante.
(define salarioBase 120)

(define (calcularSalario horasTrabajadas)
(* salarioBase horasTrabajadas))

Si queremos saber el sueldo que le corresponde a Andrea llamamos a la función de la siguiente manera:
==>(calcularSalario 81)

9720

Veamos como funciona internamente nuestro programa para asi comprenderlo mejor:

(calcularSalario 81) -> llamamos a la función con 81 como parámetro de entrada

(* salarioBase 81) -> Scheme busca en el cuerpo de la función los parámetros formales y los reemplaza por los parámetros reales (es decir, reemplazo horasTrabajadas por el valor con el que fue llamada la función)

(* 120 81) -> Scheme busca el valor de las variables que han sido definidas anteriormnte y las reemplaza (cambia salarioBase por 120)

(9720) -> Se realiza la operación y produce el resultado

1.3.1. Ejercicios

Decir cuales de las siguientes expresiones estan bien escritas cuales no y por qué

1. (8 + 4)
2. ((+ 4 2)5)
3. (define (void t)
   (* t 20))
4. (define (areaCuadrado l)
   (* l l))
5. (* (- (* 4 3) (+ 5 2)) 2)

1.4. Expresiones

1.4.1. Simples

Las expresiones más simples que se pueden escribir en Scheme son los números, como ya sabemos, al escribir un número el intérprete nos responde con el mismo número:
En Scheme las expresiones simples también son llamadas "átomos"debido a que es la unidad más pequeña del lenguaje con un valor asociado.
Son ejemplos de expresiones simples: 9, 8.5, 'symbol.

1.4.2. Compuestas

Expresiones compuestas son todas aquella que se sacan de unir 2 o más expresiones simples por medio de un operador, por ejemplo una operación es una expresión compuesta

==>(* 20 8)
160

Asi mismo:
La Expresión 1:

$$(- 5 (* (+ 2 3) (/ 32 4)))$$

esta formada por varias expresiones compuestas

$$\overbrace{(- 5 \underbrace{(* (+ 2 3) (/ 32 4))}_{Expresión\ 2})}^{Esta\ es\ una\ expresión\ compuesta}$$

$$\overbrace{(* (+ 2 3) \underbrace{(/ 32 4))}_{Expresión\ 3})}^{La\ expresión\ 2\ también\ es\ una\ expresión\ compuesta}$$

$$\overbrace{(/ \underbrace{32}_{Expresión\ Simple} 4)}^{La\ expresión\ 3\ esta\ compuesta\ por\ expresiones\ simples}$$

1.4.3. Aritméticas

Son aquellas que constan de operadores aritméticos (+, -, *, /,) y valores numéricos, como resultado devuelven un valor numerico.
Ejemplo:
==>(+ (* 12 5) 9)
69

1.4.4. Relacionales

Las expresiones relacionales son las compuestas por operadores de relacion, ya sea de orden o igualdad. Estos operadores son: mayor que, menor que, mayor o igual que, menor o igual que, igual que. (>, <, >=, <=, =).
Ejemplo:
==>(> 20 9)
;pregunta si 20 es mayor que 9
true

1.4.5. Lógicas

Son aquellas que tienen un valor de verdad (valores booleanos), estos son verdadero o falso. Scheme reconoce verdadero como #t o true y falso como #f o false.
Algunos de los operadores que podemos utilizar son boolean? para preguntar si la siguiente expresion es un valor de verdad; not, and, or.
Ejemplos:
==>(boolean? 9)
;pregunta si 9 es un valor de verdad

false

==>(number? 9)
true

==>(not #t)
#f

Estas son algunas funciones básicas del Álgebra booleana :

P	Q	P and Q
F	F	F
F	V	F
V	F	F
V	V	V

P	Q	P or Q
F	F	F
F	V	V
V	F	V
V	V	V

P	not P
F	V
V	F

1.5. Modularidad

Divide y Conquistarás

Una de las características de tener un buen estilo de programación es la modularidad. Esto significa dividir problema en problemas mas pequeños; de esa manera tenemos 2 ventajas:
Primero, tenemos que solucionar problemas más pequeños uno a la vez, lo que nos ayuda a entender el problema por partes, y reduce la cantidad de codigo que tenemos que entender a la vez, es decir la complejidad del codigo.
Segundo, la reutilización del código. Si las funciones que usamos tienen propósitos más generales se nos facilita reutilizalas. Por ejemplo, si escribimos todo un programa que calcule las notas finales de los estudiantes de cierta materia en un solo bloque es posible que ese programa solo nos sirva para esa materia especificamente. En cambio, si construimos el programa en pequeños bloques (por ejemplo, una funcion que reciba la nota de cada parcial y el porcentaje que le es asignado y otra que sume los totales, y por ultimo una que diga si la materia fue aprobada o no) nuestro programa sería útil no sólo para esa materia, sino para otra y cuando la cantidad de notas o el porcentaje asignado por el profesor para cada una de las evaluaciones cambie, solo tendriamos que cambiar una o dos lineas en el código para actualizar el programa.
Cuando trabajamos con módulos la función principal es aquella que ejecutamos para que produzca los resultados, y para poder hacer su trabajo esta llama a otras funciones encargadas de hacer tareas más pequeñas, por esta razón se les llama funciones auxiliares.
En nuestro ejemplo de la sección 1.1.1, pág 9.
Sin modularidad

```
(define (cambiarAPesos cantidadDeLaMoneda valorDeLaMoneda impuesto)
  (-
  (* cantidadDeLaMoneda valorDeLaMoneda)
    (/ (* (* cantidadDeLaMoneda valorDeLaMoneda) impuesto)100)))
```

Con modularidad

```
(define (cambiarAPesos cantidadDeLaMoneda valorDeLaMoneda impuesto)
(- (calculaCambioBruto cantidadDeLaMoneda valorDeLaMoneda)
  (calculaImpuesto cantidadDeLaMoneda valorDeLaMoneda impuesto)))

(define (calculaCambioBruto cantidadDeLaMoneda valorDeLaMoneda)
  (* cantidadDeLaMoneda valorDeLaMoneda))

(define (calculaImpuesto cantidadDeLaMoneda valorDeLaMoneda impuesto)
  (/ (* (calculaCambioBruto cantidadDeLaMoneda valorDeLaMoneda) impuesto) 100))
```

Es difícil con lo que se ha aprendido hasta ahora ver las ventajas de la modularidad, y lo práctico que es hacer nuestros programas por partes, pero a medida que vamos haciendo programas más extensos y que usen otras cosas que no hemos aprendido hasta ahora nos daremos cuenta de estas ventajas, y notaremos que algunos problemas son incluso imposibles de solucionar si no los dividimos en partes.

A continuación se muestra un ejemplo en el que se resuelve un problema con y sin el uso de modularidad; es recomendable que después de haber visto listas, condicionales, recursión y el uso de local se analice este problema y la forma como se ha resuelto para hacer así más evidente la necesidad de dividir los problemas.

El problema consiste en sacar el promedio de una lista de números, pero desconocemos la longitud de la lista. Las 3 funciones que están a continuacion resuelven el mismo problema, pero la primera es netamente recursiva; la segunda usa una función auxiliar y la tercera resuelve el problema usando modularidad.

Lo que debemos hacer es sumar cada uno de los números de la lista y dividirlos entre la cantidad de elementos que hay en esa lista.

```
(define lista1 (list 50 15 10))

(define (promedio1 aln)
(local(
  (define (promedio-aux aln contadorElem contadorSumador)
    ( cond
     [(empty? aln) (/ contadorSumador contadorElem)]
     [else ( promedio-aux (rest aln) (+ contadorElem 1) (+ contadorSumador (first aln))) ]
    )
  )
)(promedio-aux lista1 0 0)))


(promedio1 lista1)
```

```
(define (promedio2 aln)
(local (
        ;;calculaElementos cuenta los elementos en la lista
        ;;y luego llama a la función calculaPromedio
        (define (calculaElementos aln aln2 contador)
          ( cond
            [(empty? aln) (calculaPromedio aln2 0 contador) ]
            [ else (calculaElementos (rest aln) aln2 (+ contador 1))]
          )
        )
        ;;calculaPromedio suma los numeros de la lista y luego los divide por la cantidad
        ;;de elementos
        (define (calculaPromedio aln sumador contador)
          (cond
            [(empty? aln) (/ sumador contador) ]
            [ else ( calculaPromedio (rest aln) (+ sumador (first aln)) contador)]
          )
        )
)(calculaElementos aln aln 0)))

(promedio2 lista1)




;;Esta función es modular, la función principal divide el resultado de
;;sumar los numeros de una lista entre la cantidad de números que hay en la lista
(define (promedio3 aln)
  (/ (sumaListaNumeros aln) (cuantoshay? aln) ))


;;Esta función cuenta los elementos de una lista
( define ( cuantoshay? aln)
  ( local (
          ( define ( cuantoshayaux? aln1 acumulador)
            ( cond
              [(empty? aln1) acumulador ]
              [ else (cuantoshayaux? ( rest aln1) ( + 1 acumulador))]
            )
          )
)( cuantoshayaux? aln 0)))


;;Esta función suma una lista de números
(define (sumaListaNumeros aln)
  (cond
     [(empty? aln) 0]
     [else (+ (first aln)(sumaLista (rest aln)))])
  )
```

(promedio3 lista1)

Vimos que resulta más fácil solucionar problemas más pequeños, el código de la tercera función (promedio3) es más fácil de entender que los otros; y además es reutilizable. Ejemplo:

Para sacar las calificaciones de todos los cursos que dicta un profesor se pide desarrollar una función que reciba dos parámetros:

Una lista de porcentajes (indica cuanto vale cada nota) ej: (list 10 40 50) nos indica que la primera nota vale el 10 % la segunda 40 % y la tercera el 50 % de la nota final. Y una lista de calificaciones.
Además debemos verificar que las listas que nos den cumplan ciertas condiciones; las listas deben tener igual longitud pues no tiene sentido que hallan más o menos calificaciones que evaluaciones realizadas, y la lista de porcentajes debe sumar exactamente el 100 % de la nota final.

```
(define (verificarNota listaPorcentajes listaCalificaciones)
;;verifica que los parámetros que nos dieron sean correctos
  (cond
    [(not(=(sumaListaNumeros listaPorcentajes) 100)) -1]
    ;; comprueba si la lista de procentajes suma el 100 %, en caso contrario devuelve -1
    [else
     (cond
       [(not (= (cuantoshay? listaPorcentajes) (cuantoshay? listaCalificaciones))) -1]
       ;; comprueba que las listas tengan la misma longitud
       [else (sacarNota listaPorcentajes listaCalificaciones)])]))
       ;;si los parámetros son correctos, llama a la otra función que calcula la nota final


(define (sacarNota listaPorcentajes listaCalificaciones)
;;saca la nota final dada la lista de porcentajes y la lista de notas
  (cond
    [(empty? listaPorcentajes) 0]
    [else
     (+ ( / (* (first listaPorcentajes) (first listaCalificaciones)) 100 )
       (sacarNota (rest listaPorcentajes) (rest listaCalificaciones)))
     ]))

(define listaPorcen (list 50 30 20 ))
(define listaNotas (list 5 3 5 ))
(verificarNota listaPorcen listaNotas)
```

Podemos observar que para resolver esta problema usamos funciones que ya

estaban definidas anteriormente (**cuantoshay?** para contar los elementos de la lista y **sumaListaNumeros** para verificar que entre todos los porcentajes sumen 100 %), de esta manera hemos ahorrado tiempo en escribir código, y hemos visto otra de las ventajas de la modularidad.

1.6. Prueba y depuración

Depurar es el proceso en el cual el programador corrige el código. Podemos darnos cuenta que algo esta mal en nuestro programa cuando al compilarlo o al usarlo nos generan mensajes de error, o cuando no produce los resultados esperados Es importante darse cuenta que la corrección de un programa tiene 3 aspectos:

- Sintaxis ¿El computador entiende lo que yo le quiero decir?
- Semántica ¿Siempre produce los resultados que yo espero?
- Pragmática ¿El programa es fácil de entender y de usar?

Veremos algunos ejemplos de errores comunes:

1.6.1. Sintaxis

Algunos errores se producen por la forma de escribir, por ejemplo en la siguiente función:

```
(define (Multip valor1 valor2)
(* valor1 valor2))

(multip 3 3)
```

Al ejecutar el código nos va a mostrar un error:

reference to undefined identifier: multip

La razón de esto es que Scheme es CASE SENSITIVE, es decir, reconoce la diferencia entre maúsculas y minúsculas. Por lo tanto Perro es diferente de perro. Lo que debemos hacer es cambiar la palabra Multip (por cuestiones de estilo) y ponerla en minúsculas.

En la función calcular

```
(define (calcular v1 v2 v3)
(+ (costo1 v1 v2) costo2 v2 v3)))

(define (costo1 v1 v2)
(*v1 v2))

(define (costo2 v2 v3)
(+ v2(* v3 15)))
```

Hay un error de sintaxis, hace falta un paréntesis en la segunda línea, justo antes de llamar a la función auxiliar costo2. Las dos primeras lineas deben quedar así:
(define (calcular v1 v2 v3)
(+ (costo1 v1 v2) (costo2 v2 v3)))

Los errores de sintaxis usualmente son fáciles de corregir, puesto que el mismo computador nos señala en dónde estuvo el error, si el error se nos hace difícil de localizar o no es tan obvio procedemos de la manera siguiente:
Buscamos el error en la linea que estamos trabajando, si alli no lo encontramos pasamos a la linea inmediatamente anterior y así sucesivamente; es posible que se nos halla olvidado por ejemplo cerrar un paréntesis en medio de una función y el computador nos va a señalar el error 2 o 3 lineas más abajo, al finalizar la definición del procedimiento. Si no podemos encontrar el error alli debemos probar el programa por partes, es decir, encerrar en comentarios parte del programa y probar las funciones auxiliares una por una para ver cuál de ellas genera el error. Es una buena práctica ir probando el código a medida que se escribe para darnos cuenta de los errores que vamos cometiendo y asi depurarlos más fácil. Podemos empezar con una función básica y cuando ésta se ejecute correctamente ir definiendo otros procedimientos, y luego volver a correr el programa para ver si la porción de código que agregamos funciona ben. No debemos seguir escribiendo más código hasta estar seguros de que lo que tenemos en el momento funciona bien.

1.6.2. Semántica

Los errores de semántica son los más difíciles de detectar y corregir. Algunas veces no muestran errores al momento de ejecutar el programa, de allí la dificultad de la depuración. Estos errores se producen cuando escribimos un código que tiene sentido en el aspecto lingüístico pero le damos al computador una orden incorrecta
Supongamos que nosotros definimos la variable a como 5:
(define a 5)

y definimos un función que hace una suma :

(define (suma a b)
(+ a b))

Si hacemos un llamado a la función:

(suma 2 3)
Recibimos como respuesta *5*

Es posible pensar que como anteriormente definimos la variable **a** como 5 al llamar a la función suma esta va a reemplazar el valor de **a** en 5 y de **b** en 3, produciendo 8. Pero tenemos que tener en cuenta que **a** es un parámetro formal de la función, por lo tanto Scheme reemplazará el valor en la función por el parámetro dado, o sea que usará el 2 en lugar del 5.
Vemos pues que aunque este bien escrito el código, no hace lo que se pensaba. La forma correcta para que produciera el resultado usando el número 5 es:

(suma a 3)
y recibimos como respuesta *8*

1.6.3. Pragmática

Es posible que un programa funcione bien pero este mal hecho porque es demasiado difícil de entender, de leer, de usar o de modificar. Por ejemplo:
En un club social los clientes han propuesto dar una fiesta al final de mes a todos aquellos que cumplen años, para esto han aceptado que se les cargue el costo de la fiesta al recibo de la mensualidad; pero hay que tener en cuenta que los homenajeados no deben pagar por la fiesta.

(define grt 187)(define (laVafNMma apergt bioe58 Gttb) (+ (rer grt bioe58 Gttb) apergt))
(define(rer fr gh Gttb) (/ Gttb (- fr gh)))

Este es un código que funciona correctamente, si lo ponemos en Scheme y le damos datos de entrada producirá un resultado sin sacar ningún error; sin embargo es un código muy confuso y no se entiende lo que hace, lo más seguro es que después de un tiempo ni siquiera la misma persona que lo escribió se dé cuenta de la intención que perseguía al hacer esa función; por lo tanto ese código no podrá ser reutilizado y resulta más práctico volverlo a hacer que tratar de entender lo que ya se hizo.

Vamos a corregir el código:

```
;; se supone aceptaron dar un regalo mensual a aquellos que cumplen años,
;;además deben pagar la mensualidad
(define numClientes 15)
(define (valorAPagar mensualidad clientesQueCumplenAnnos cantidad)
  (+
    (calcularCosto numClientes clientesQueCumplenAnnos cantidad)
     mensualidad))
(define(calcularCosto totalClientes cumplen dinero)
  ( / dinero (- totalClientes cumplen )) )
```

El código bien explicado y comentado es lo siguiente:

```
(define numClientes 15)
;; definimos el numero de clientes del club
;; la función valorAPagar recibe la mensualidad, el número de clientes que
;; cumplen años y la cantidad de dinero que se estima será gastada en la fiesta
;; la función debe devolver el valor a pagar por cada
;; cliente en su factura.
(define (valorAPagar mensualidad clientesQueCumplenAnnos cantidad)
(+ (calcularCosto numClientes clientesQueCumplenAnnos cantidad) mensualidad))
;; la función calcularCosto recibe la cantidad total de clientes,
;;el numero de clientes que cumplen años en ese mes y la cantidad
;; estimada de dinero que se gastará en la fiesta
```

```
;; la funcion devuelve el aporte extra que debe dar cada cliente
;; para cubrir los costos de la fiesta
(define(calcularCosto totalClientes cumplen dinero)
( / dinero (- totalClientes cumplen )) )
```

1.7. Problemas

1. Hacer una función que nos de el resultado de la sumatoria de n, recordar esta fórmula:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

```
(define (sumatoria n)
  ( / (* n (+ 1 n)) 2 ))

(sumatoria 5)
```
==>15

2. Hacer una función que simule la suma de 2 números fraccionarios, dados así:

$$\frac{a}{b} + \frac{c}{d} = \frac{(a \times d) + (c \times b)}{b \times d}$$

```
;suma a/b + c/d
(define (sumafraccionario a b c d)
  (/ (+ (* a d) (* c b) ) (* b d)))

;;para sumar 1/2 + 2/5
(sumafraccionario 1 2 2 5)
```
==>9/10

3. Para el siguiente ejemplo necesitamos saber manejar listas y condiciones: Construya una función que pase un número natural a su notación binaria

```
(define (pasabinario a)
  (cond  [(> a 0) (cons (remainder a 2) (pasabinario (quotient a 2)))]
         [else empty]))

(pasabinario 9)
```
==>(list 1 0 0 1)