

Introducción

Todos hemos tenido contacto con los números. De igual manera hemos tenido contacto con fórmulas que involucran números y símbolos. Parece natural iniciar el proceso de aprendizaje en una área en la cual se posee un conocimiento básico.

El presente capítulo demuestra como es posible por medio de Scheme escribir procesos que realicen computaciones numéricas. En especial fórmulas recursivas, fórmulas que se definen en términos de sí mismas.

El poder de este tipo de descripciones consiste fundamentalmente en el hecho de poder poder descomponer un problema en un nuevo problema que sea matemáticamente más sencillo que el anterior. Se desea hacer un proceso de reducción continua hasta que se llegue a un caso inicial sencillo de resolver.

La función sucesor

La función sucesor es una función muy sencilla. Lo que se desea es enviar un número entero y recibir como respuesta el sucesor inmediato de dicho número.

De esta forma, la función que se denominará (*suc num*) y debe producir los siguientes resultados:

> (*suc 0*)

1

> (*suc 5*)

6

```
> (suc 10)  
11
```

Tal como se muestra en los ejemplos el proceso consiste en tomar un número x , y producir como resultado el valor de $x+1$. Observe que además de la función se han incluido comentarios para facilitar el entendimiento del algoritmo que se desea programar. De esta forma se construye la siguiente función

```
;; Función para obtener el sucesor de un número  
;; Precondiciones  
;; x : número entero  
;;  
(define (suc x)  
  (+ x 1))
```

Si bien la función anterior funciona perfectamente, es importante tratar de usar variables dentro de las funciones que sean lo más significativas posibles. Por ello se prefiere el estilo que se muestra a continuación:

```
;; Función para obtener el sucesor de un número  
;; Precondiciones  
;; num : número entero  
;;  
(define (suc num)  
  (+ num 1))
```

La función predecesor

La función de predecesor realiza la función inversa a la de sucesor. Para un valor de x , debe construir el valor de $x-1$. Se espera que opere de la siguiente manera:

```
> (pred 1)  
0  
  
> (pred 5)  
4  
  
> (pred 10)  
9
```

Esta función se puede escribir de la siguiente manera:

- Función para obtener el predecesor de un número
- Precondiciones
- num : número entero

```
(define (pred num)
  (- num 1))
```

Elevar al cuadrado un número

Esta función recibe un número x y devuelve el valor de dicho número al cuadrado. Debe producir los siguientes resultados.

- (cuadrado 1)
1
- (cuadrado 2)
4
- (cuadrado 3)
9
- (cuadrado 4)
16

Se puede construir esta función de la siguiente manera:

- Función que eleva el valor de x al cuadrado
- Precondiciones
- x : número real

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

Elevar un número a una potencia

Una de las funciones más comunes que realiza cualquier calculadora consiste en elevar un número a una potencia. Eso significa multiplicar ese número por sí

misma cierta cantidad de veces. En este ejemplo la base puede ser cualquier número real y la potencia debe ser un número entero mayor o igual a cero.

El algoritmo descrito anteriormente puede escribirse matemáticamente mediante la fórmula:

$$x^0 = 1$$

$$x^n = x * x^{(n - 1)}; \text{ } n \text{ entero y } n > 0$$

Se establece un proceso de cálculo, donde un problema se descompone en otro problema más pequeño hasta que se llega a una solución final. Por ejemplo si se desea calcular el valor de 2 elevado a la 4 se tiene que:

$$\begin{aligned} 2^4 &= 2 * (2^3) \\ &= 2 * (2 * 2^2) \\ &= 2 * (2 * (2 * 2^1)) \\ &= 2 * (2 * (2 * (2 * (2 * 2^0)))) \\ &= 2 * (2 * (2 * (2 * (2 * 1)))) \\ &= 2 * (2 * (2 * 2)) \\ &= 2 * (2 * 4) \\ &= 2 * 8 \\ &= 16 \end{aligned}$$

Al utilizar la función con otros valores numéricos debe producir los siguientes resultados:

> (elevar 2 2)

4

> (elevar 2 3)

8

> (elevar 2 4)

16

> (elevar 3 2)

9

* (elevar 3 3)

27

* (elevar 3 4)

81

* (elevar 5 2)

25

* (elevar 2.11 2)

4.4521

* (elevar 2.11 3)

9.393931

Para construir esta función se implementará el siguiente algoritmo, elevar un número (que denominaremos x) a una potencia (que se denominará n) es equivalente a multiplicar x por el resultado de elevar n a una potencia de n-1. Se debe proceder así hasta que el valor de n sea 0, en cuyo caso n elevado a la 0 da como resultado el valor de 1. Básicamente se está siguiendo el proceso descrito por la fórmula matemática.

* (elevar 2 4)

(* 2 (elevar 2 3))

(* 2 (* 2 (elevar 2 2)))

(* 2 (* 2 (* 2 (elevar 2 1)))))

(* 2 (* 2 (* 2 (* 2 (elevar 2 0))))))

(* 2 (* 2 (* 2 (* 2 1)))))

(* 2 (* 2 (* 2 2)))

(* 2 (* 2 4))

(* 2 8)

16

Dicha fórmula se puede escribir mediante el siguiente algoritmo:

□ Función para elevar un número a una potencia

□ Precondiciones

□ x: número real

□ n: número entero y $n \geq 0$

□

```
(define (elevar x n)
  (cond ((zero? n)
         1)
        (else
         (* x (elevar x (- n 1))))))
```

Sumatoria de 0 hasta n

Inicialmente se construirá una función que realice la sumatoria de 0 hasta un número n. Donde n debe ser un número entero mayor o igual a cero. Se trata de construir la fórmula conocida como:

$$\sum_{i=0}^n i = 0 + 1 + 2 + \dots + n$$

Si se desea sumar de 0 hasta 4 se debe generar la siguiente suma:

$$0 + 1 + 2 + 3 + 4 = 10$$

Sumar de 0 hasta 4 es igual que sumar de 4 hasta 0, por lo que se puede generar también la siguiente suma:

$$4 + 3 + 2 + 1 + 0 = 10$$

Se desea construir una función que se llamará (sumar-hasta num), la cual debe producir los siguientes resultados:

```
> (sumar-hasta 0)
0
> (sumar-hasta 1)
1
> (sumar-hasta 2)
3
> (sumar-hasta 3)
6
```

18 MAR 2009

> (sumar-hasta 4)

10

> (sumar-hasta 10)

55

> (sumar-hasta 100)

5050

005.369
H479i²

84529

Para construir dicha función es importante notar que sumar hasta n, es igual que sumar hasta $n-1$ y después a eso sumarle el valor de n. Es decir, para sumar hasta 4, basta que se tenga el valor de sumar de 3 hasta 0 y a ese valor sumarle 4. Este proceso continua hasta que se solicite sumar de 0 hasta 0, cuyo resultado es 0.

> (sumar-hasta 4)

(+ 4 (sumar-hasta 3))

(+ 4 (+ 3 (sumar-hasta 2)))

(+ 4 (+ 3 (+ 2 (sumar-hasta 1))))

(+ 4 (+ 3 (+ 2 (+ 1 (sumar-hasta 0)))))

(+ 4 (+ 3 (+ 2 (+ 1 0)))))

(+ 4 (+ 3 (+ 2 1))))

(+ 4 (+ 3 3))

(+ 4 6)

10

A continuación se presenta el código de dicha función.

;; Función que suma hasta num

;; Precondiciones

;; num: número entero, num >= 0

;;

(define (sumar-hasta num)

(cond ((zero? num)

0)

(else

(+ num (sumar-hasta (- num 1))))))

Sumatoria de cuadrados

Una función similar a la sumatoria de 0 hasta n consiste en elevar al cuadrado un conjunto de números y luego sumarlos. Se desea calcular la suma de los cuadrados de un conjunto de números de 0 hasta n, donde n es un número entero mayor o igual a cero. Esta función se puede expresar mediante la fórmula:

$$\sum_{i=0}^n i^2 = 0^2 + 1^2 + 2^2 + \dots + n$$

Al igual que en el caso anterior, si se desea sumar los cuadrados de los números de 0 a 4 se realizará la siguiente suma:

$$4^2 + 3^2 + 2^2 + 1^2 + 0^2 = 30$$

Se desea construir una función que tenga el siguiente comportamiento:

> (sumar-cuad 1)

1

> (sumar-cuad 2)

5

> (sumar-cuad 3)

14

> (sumar-cuad 4)

30

> (sumar-cuad 10)

385

> (sumar-cuad 100)

338350

De esta manera, se puede modificar el programa anterior para que en lugar de sumar los números de 0 hasta n, los eleve al cuadrado y luego los sume.

> (sumar-cuad 4)

(+ 16 (sumar-cuad 3))

(+ 16 (+ 9 (sumar-cuad 2))))

```
(+ 16 (+ 9 (+ 4 (sumar-cuad 1))))
(+ 16 (+ 9 (+ 4 (+ 1 (sumar-cuad 0)))))
(+ 16 (+ 9 (+ 4 (+ 1 0))))
(+ 16 (+ 9 (+ 4 1)))
(+ 16 (+ 9 5))
(+ 16 14)
30
```



Se obtiene entonces el siguiente programa:

```
;; Función que suma cuadrados de 0 hasta num
;; Precondiciones
;; num: número entero, num >= 0
;;
(define (sumar-cuad num)
  (cond ((zero? num)
         0)
        (else
         (+ (* num num) (sumar-cuad (- num 1))))))
```

Es importante mencionar que es posible utilizar el código que se ha escrito anteriormente. Una función puede invocar a otra función previamente definida. De esta manera, el programa anterior podría reescribirse como:

```
;; Función que suma cuadrados de 0 hasta num
;; Precondiciones
;; num: número entero, num >= 0
;;
(define (sumar-cuad num)
  (cond ((zero? num)
         0)
        (else
         (+ (cuadrado num) (sumar-cuad (pred 1))))))
```

Sumatoria de cocientes

El ejemplo que se muestra a continuación es un caso clásico que se utiliza en la enseñanza de la inducción matemática.

$$\sum_{i=1}^n \frac{i}{i*(i+1)} = \frac{1}{1*2} + \frac{1}{2*3} + \dots + \frac{1}{n*(n+1)}$$

La suma de los primeros cuatro términos de esta suma es:

En general se espera que esta función produzca los siguientes resultados:

$$0 + \frac{1}{1*2} + \frac{1}{2*3} + \frac{1}{3*4} + \frac{1}{4*5} = \frac{4}{5}$$

O bien de forma equivalente:

$$\frac{1}{4*5} + \frac{1}{3*4} + \frac{1}{2*3} + \frac{1}{1*2} + 0 = \frac{4}{5}$$

Que al operar los cocientes se convierte en:

$$\frac{1}{20} + \frac{1}{12} + \frac{1}{6} + \frac{1}{2} + 0 = \frac{4}{5}$$

Otros de los resultados que debe producir son:

> (sumar-coc 0)

0

> (sumar-coc 1)

1/2

> (sumar-coc 2)

2/3

> (sumar-coc 3)

3/4

> (sumar-coc 10)

10/11

> (sumar-coc 100)

100/101

Si se desea que el valor se retorne con punto en lugar del formato de fracción, bastaría agregarlo al valor que se pregunta:

```
> (sumar-coc 0.0)  
0  
> (sumar-coc 1.0)  
0.5  
> (sumar-coc 2.0)  
0.666666666666666...  
> (sumar-coc 3.0)  
0.75  
> (sumar-coc 10.0)  
0.9090909090909091...  
> (sumar-coc 100.0)  
0.9900990099009898...
```

Se tiene que programar el siguiente proceso:

```
> (sumar-coc 4)  
(+ 1/20 (sumar-coc 3))  
(+ 1/20 (+ 1/12 (sumar-coc 2)))  
(+ 1/20 (+ 1/12 (+ 1/6 (sumar-coc 1))))  
(+ 1/20 (+ 1/12 (+ 1/6 (+ 1/2 (sumar-coc 0)))))  
(+ 1/20 (+ 1/12 (+ 1/6 (+ 1/2 0)))))  
(+ 1/20 (+ 1/12 (+ 1/6 1/2)))  
(+ 1/20 (+ 1/12 2/3))  
(+ 1/20 3/4)  
4/5
```

El código para construir la función anterior podría escribirse así:

```
;; Suma de Cocientes  
;; Precondiciones  
;; num: número entero, num >= 1  
;;  
(define (sumar-coc num)  
  (cond ((zero? num)
```

```
0)
  (else
    (+ (/ 1 (* num (+ 1 num)))
        (sumar-coc (- num 1))))))
```

Sumatoria de un intervalo

Se desea construir una función que sume desde un valor inicial denominado ini hasta un valor final denominado fin, donde ini y fin son dos números enteros. Asimismo ini debe ser menor o igual a fin. Se debe implementar la fórmula matemática descrita por:

$$\sum_{i=ini}^{fin} i = (ini) + (ini + 1) + (ini + 2) + \dots + (fin)$$

Si se desea sumar desde 5 hasta 10 se debe generar la siguiente suma:

$$10 + 9 + 8 + 7 + 6 + 5 = 45$$

De esta forma, se desea construir una función que se llamará (sumar-intervalo ini fin), la cual debe producir los siguientes resultados:

```
> (sumar-intervalo 0 10)
55
> (sumar-intervalo 1 10)
55
> (sumar-intervalo 2 10)
54
> (sumar-intervalo 3 10)
52
> (sumar-intervalo 5 10)
45
> (sumar-intervalo 5 20)
200
```

Para construir esta función se iniciará sumando el valor de ini, con el valor de $ini + 1$ y así sucesivamente hasta que se alcance el valor de fin. Sumar de ini hasta fin, es lo mismo que sumarle ini al valor obtenido de sumar de $ini + 1$ hasta fin. Finalmente sumar de fin hasta fin, tiene un valor de fin.

```

• (sumar-intervalo 5 10)
  (+ 5 (sumar-intervalo 6 9))
  (+ 5 (+ 6 (sumar-intervalo 7 10)))
  (+ 5 (+ 6 (+ 7 (sumar-intervalo 8 10))))
  (+ 5 (+ 6 (+ 7 (+ 8 (sumar-intervalo 9 10)))))
  (+ 5 (+ 6 (+ 7 (+ 8 (+ 9 (sumar-intervalo 10 10))))))
  (+ 5 (+ 6 (+ 7 (+ 8 (+ 9 10))))))
  (+ 5 (+ 6 (+ 7 (+ 8 19))))))
  (+ 5 (+ 6 (+ 7 27))))))
  (+ 5 (+ 6 34)))
  (+ 5 40)
  45

;; Suma un intervalo de ini a fin
;; Precondiciones
;; ini y fin números enteros
;; ini <= fin
;;
(define (sumar-intervalo ini fin)
  (cond ((equal? ini fin)
         fin)
        (else
         (+ ini
            (sumar-intervalo (+ 1 ini) fin))))))
```

Sumatoria aplicando una función

Se desea escribir el código para una función reciba otra función denominada `fun` y la aplica a cada uno de sus elementos. Los elementos son números enteros que van desde un valor inicial denominado `ini` hasta un valor final denominado `fin`.

Se desea programar la fórmula definida por:

$$\sum_{i=ini}^{fin} fun(i) = fun(ini) + fun(ini+1) + \dots + fun(fin)$$

Se desea que dicha función produzca los siguientes resultados:

> (sumarf suc 0 4)

15

> (sumarf suc 0 9)

55

> (sumarf cuadrado 0 4)

30

> (sumarf cuadrado 0 100)

338350

El comportamiento de esta función debe ser el siguiente:

```
> (sumarf cuadrado 0 4)
(+ 0 (sumarf cuadrado 1 4))
(+ 0 (+ 1 (sumarf cuadrado 2 4)))
(+ 0 (+ 1 (+ 4 (sumarf cuadrado 3 4))))
(+ 0 (+ 1 (+ 4 (+ 9 (sumarf cuadrado 4 4)))))
(+ 0 (+ 1 (+ 4 (+ 9 16))))
(+ 0 (+ 1 (+ 4 25)))
(+ 0 (+ 1 29))
(+ 0 30)
30
```

Dicha función se puede construir de la siguiente manera:

- :: Suma un intervalo y
- :: aplica una función a los elementos
- :: Precondiciones
- :: fun: una función lambda válida
- :: ini, fin: números enteros
- :: ini <= fin

```
(define (sumarf fun ini fin)
  (cond ((equal? ini fin)
         (fun fin))
        (else
         (+ (fun ini)
            (sumarf fun (+ 1 ini) fin))))))
```

La función factorial

El simbolo “!” se llama factorial. La función factorial se define como:

$$0! = 1$$

$$n! = n \cdot (n-1)!$$

De esta forma se puede obtener que:

$$0! = 1$$

$$1! = (1) \cdot (0!) = (1) \cdot (1) = 1$$

$$2! = (2) \cdot (1!) = (2) \cdot (1) = 2$$

$$3! = (3) \cdot (2!) = (3) \cdot (2) = 6$$

$$4! = (4) \cdot (3!) = (4) \cdot (6) = 24$$

$$5! = (5) \cdot (4!) = (5) \cdot (24) = 120$$

Por lo tanto si se desea calcular $4!$ es necesario calcular un conjunto de valores anteriores. Por ejemplo:

$$\begin{aligned} 4! &= 4 * 3! \\ &= 4 * 3 * 2! \\ &= 4 * 3 * 2 * 1! \\ &= 4 * 3 * 2 * 1 * 0! \\ &= 4 * 3 * 2 * 1 * 1 \\ &= 4 * 3 * 2 * 1 \\ &= 4 * 3 * 2 \end{aligned}$$

```
= 4 * 6
```

```
= 24
```

Se desea entonces construir una función que reciba un número entero positivo y genere el valor factorial para ese número. El comportamiento de la función debe ser:

```
> (fact 0)
```

```
1
```

```
> (fact 1)
```

```
1
```

```
> (fact 2)
```

```
2
```

```
> (fact 3)
```

```
6
```

```
> (fact 4)
```

```
24
```

```
> (fact 5)
```

```
120
```

Para calcular el factorial de num es suficiente multiplicar el valor de num por el factorial de num-1. Se debe continuar así sucesivamente hasta llegar al factorial de 0 cuyo valor es 1.

```
> (fact 4)
(* 4 (fact 3))
(* 4 (* 3 (fact 2)))
(* 4 (* 3 (* 2 (fact 1))))
(* 4 (* 3 (* 2 (* 1 (fact 0)))))
(* 4 (* 3 (* 2 (* 1 1))))
(* 4 (* 3 (* 2 1)))
(* 4 (* 3 2))
(* 4 6)
24
```

```
:: Calcula el factorial de num
```

```
:: Precondiciones
```

```
;; num: número entero, num >= 0
;;
(define (fact num)
  (cond ((zero? num)
         1)
        (else
         (* num (fact (- num 1))))))
```

Los números de Fibonacci

Los números de Fibonacci son un ejemplo de una sucesión infinita, donde a cada número entero positivo n , le corresponde un número $f(n)$. Esta relación entre n y $f(n)$ está descrita por:

$$\begin{aligned} \text{fib}(0) &= 1 \\ \text{fib}(1) &= 1 \\ \text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2) \end{aligned}$$

Calculemos algunos valores de la serie de fibonacci:

$$\begin{aligned} \text{fib}(0) &= 1 \\ \text{fib}(1) &= 1 \\ \text{fib}(2) &= \text{fib}(1) + \text{fib}(0) = 1 + 1 = 2 \\ \text{fib}(3) &= \text{fib}(2) + \text{fib}(1) = 2 + 1 = 3 \\ \text{fib}(4) &= \text{fib}(3) + \text{fib}(2) = 3 + 2 = 5 \\ \text{fib}(5) &= \text{fib}(4) + \text{fib}(3) = 5 + 3 = 8 \\ \text{fib}(6) &= \text{fib}(5) + \text{fib}(4) = 8 + 5 = 13 \\ \text{fib}(7) &= \text{fib}(6) + \text{fib}(5) = 13 + 8 = 21 \\ \text{fib}(8) &= \text{fib}(7) + \text{fib}(6) = 21 + 13 = 34 \\ \text{fib}(9) &= \text{fib}(8) + \text{fib}(7) = 34 + 21 = 55 \\ \text{fib}(10) &= \text{fib}(9) + \text{fib}(8) = 55 + 34 = 89 \end{aligned}$$

Para calcular el fibonacci de 4 se requiere calcular un conjunto de valores anteriores:

$$\begin{aligned}\text{fib}(4) &= \text{fib}(3) + \text{fib}(2) \\&= (\text{fib}(2) + \text{fib}(1)) + (\text{fib}(1) + \text{fib}(0)) \\&= (\text{fib}(1) + \text{fib}(0) + \text{fib}(1)) + (\text{fib}(1) + \text{fib}(0)) \\&= (1 + 1 + 1) + (1 + 1) \\&= 3 + 2 \\&= 5\end{aligned}$$

Se construirá una función que exhiba el siguiente comportamiento:

> (fib 0)

1

> (fib 0)

1

> (fib 1)

1

> (fib 2)

2

> (fib 3)

3

> (fib 4)

5

> (fib 5)

8

> (fib 6)

13

> (fib 7)

21

> (fib 8)

34

> (fib 9)

55

> (fib 10)

89

Que al programar la función se debe convertir en:

- * (fib 4)
 - (+ (fib 3) (fib 2))
 - (+ (+ (fib 2) (fib 1)) (+ (fib 1) (fib 0)))
 - (+ (+ (+ (fib 1) (fib 0)) (fib 1)) (+ (fib 1) (fib 0)))
 - (+ (+ (+ 1 1) 1) (+ 1 1))
 - (+ (+ 2 1) (+ 1 1))
 - (+ 3 2)

5

La función que genera los números de Fibonacci está escrita por:

- La función de Fibonacci
- Precondiciones
- num entero positivo
- num ≥ 0
-
- (define (fib num)
 (cond ((equal? num 0)
 1)
 ((equal? num 1)
 1)
 (else
 (+ (fib (- num 1))
 (fib (- num 2)))))))



Determinar si un número es primo

Un número n es primo si y solo si es divisible únicamente por 1 y por sí mismo. Un número n es divisible por un número d si su residuo es igual a 0. A continuación se construirá una función llamada (primo? num) la cual recibe un

número entero mayor o igual a 1 y produce como resultado el valor de #t si se trata de un número primo y produce #f en cualquier otro caso.

Aquí se presentan algunos ejemplos adicionales.

```
> (primo? 1)
#t

> (primo? 2)
#t

> (primo? 3)
#t

> (primo? 4)
#f

> (primo? 110)
#f

> (primo? 111)
#f

> (primo? 113)
#t
```

Se utilizará una función llamada (divisible? num den) que recibe dos argumentos y devuelve #t si el primer argumento es divisible por el segundo y #f en cualquier otro caso.

La función debe programarse de la siguiente manera. Si el número es 1, entonces es primo. Si no es uno se inicia una secuencia de búsqueda empezando en 2. Si el número es divisible por 2, no es primo. En caso contrario se prueba con 3 y así sucesivamente hasta que suceda uno de los dos casos siguientes. Si algún número lo divide no es primo. Si se llega hasta el valor del número, entonces si es primo.

```
> (primo? 1)
#t

> (primo? 7)
(primo-aux 7 2) ;; 2 no divide a 7
(primo-aux 7 3) ;; 3 no divide a 7
(primo-aux 7 4) ;; 4 no divide a 7
```

```
(primo-aux 7 5) ;; 5 no divide a 7  
(primo-aux 7 6) ;; 6 no divide a 7  
(primo-aux 7 7) ;; al llegar a 7 es primo  
#t
```

```
> (primo? 25)  
(primo-aux 25 2) ;; 2 no divide a 15  
(primo-aux 25 3) ;; 3 no divide a 25  
(primo-aux 25 4) ;; 4 no divide a 25  
(primo-aux 25 5) ;; 5 si divide a 25  
#f
```

```
|| Determina si num es divisible por div  
|| Precondiciones  
|| num: entero, num > 0  
|| div: entero, div > 0
```

```
||  
(define (divisible? num div)  
  (zero? (remainder num div)))
```

```
|| Determina si num es un número primo  
|| Precondiciones  
|| num: entero, num > 0
```

```
||  
(define (primo? num)  
  (cond ( (equal? num 1)  
          #t)  
        ( else  
          (primo?-aux num 2)))))
```

```
(define (primo?-aux num div)  
  (cond ( (equal? num div)  
          #t)  
        ( (divisible? num div)  
          #f)  
        ( else  
          (primo?-aux num (+ div 1))))))
```

Cálculo de raíces por bisección

Uno de los métodos más sencillos para encontrar la raíz cuadrada de un número es el método de bisección. Este método busca la raíz cuadrada de un número cualquiera por aproximaciones sucesivas, donde cada aproximación se consigue bisectando o bien partiendo en dos, el área de posibles soluciones.

Para mostrar este método se explicará el procedimiento que se utiliza para encontrar la raíz cuadrada de 2. Como es conocido, la raíz cuadrada de 2 es 1.414213562...

Nuestro método producirá aproximaciones sucesivas, donde se espera que cada nueva aproximación se encuentre más cerca de la respuesta correcta. Para determinar que tan cerca se encuentra una aproximación generada de la respuesta correcta es necesario definir algún criterio de decisión o bien criterio de optimalidad. Este criterio indicará cuando se debe detener el proceso, es decir, cuando se ha encontrado una aproximación que está lo suficientemente cerca de la respuesta correcta.

Un criterio de decisión que se utiliza con regularidad consiste en que la aproximación hallada y la respuesta correcta no difieran en más que una pequeña cantidad o error. Para ello se obtiene la diferencia de la respuesta y el error en valor absoluto.

$$\left| \text{Respuesta} - \text{Aproximación} \right| \leq \text{Error}$$

Como es inusual conocer la respuesta del problema que se desea resolver, puede ser necesario reescribir el criterio anterior con base en las características propias del problema. En el ejemplo propuesto, una posible reformulación del criterio de optimalidad consistiría en que la aproximación que se ha encontrado, elevada al cuadrado debe ser igual al número inicial. La aproximación se denotará mediante la variable `aprox`.

$$\left| 2 - \text{Aprox}^2 \right| \leq \text{Error}$$

Finalmente, para fijar el valor del error, si se desean 3 dígitos de exactitud, se puede solicitar que la diferencia entre los valores sea menor o igual que un error de 0.0001.

$$\left| 2 - \text{Aprox}^2 \right| \leq 0.0001$$

La primera aproximación se debe encontrar dentro del intervalo [0,2]. Pues la raíz de 2 debe ser mayor que 0 y menor que 2. Para construir las aproximaciones se toma el intervalo de las posibles soluciones y se construye una aproximación. Para la construcción de la aproximación se toma el punto medio del intervalo, en este caso, la primera aproximación X_0 estará dada por:

$$X_0 = \left(\frac{0+2}{2} \right)$$

$$X_0 = 1$$

No utiliza el criterio de optimalidad para decidir si X_0 es una buena aproximación o no. En caso que sea una buena aproximación ya se tiene una respuesta, en caso que no sea una buena aproximación se debe generar una nueva aproximación y volver a utilizar el criterio de optimalidad.

En el presente ejemplo se tiene que

$$\left| 2 - 1^2 \right| = 1$$

y se tiene que

$$1 > 0.0001$$

Como el valor de $1 > 0.0001$, se debe proceder a generar una nueva aproximación. En este caso, la nueva aproximación debe ser un número mayor que 1, pues $1^2 = 1$ y $1 < 2$. Así que la nueva aproximación se generará del intervalo [1, 2].

$$X_1 = \left(\frac{1+2}{2} \right)$$

$$X_1 = 1.5$$

Se tiene que

$$\left| 2 - 1.5^2 \right| = \left| 2 - 2.25 \right| = 0.25$$

y se tiene que

$$0.25 > 0.0001$$

Como el valor de $0.25 > 0.0001$, se debe proceder a generar una nueva aproximación. En este caso, la nueva aproximación debe ser un número menor que 1.5, pues $1.5 * 1.5 = 2.25$ y $2 < 2.25$. Así que la nueva aproximación se generará del intervalo $[1, 1.5]$.

$$X_2 = \left(\frac{1 + 1.5}{2} \right)$$

$$X_2 = 1.25$$

Se tiene que

$$\left| 2 - 1.25^2 \right| = \left| 2 - 1.5625 \right| = 0.4375$$

y se tiene que

$$0.4375 > 0.0001$$

Como el valor de $0.4375 > 0.0001$, se debe proceder a generar una nueva aproximación. En este caso, la nueva aproximación debe ser un número mayor que 1, pues $1.25 * 1.25 = 1.5625$ y $1.5625 < 2$. Así que la nueva aproximación se generará del intervalo $[1.25, 1.5]$.

$$X_3 = \left(\frac{1.25 + 1.5}{2} \right)$$

$$X_3 = 1.375$$

El algoritmo continuaría de esta manera, encontrando cada vez un intervalo más cercano al verdadero valor de la raíz cuadrada de 2.

Se construirá una función llamada raíz, que utiliza el mecanismo de la bisección. Esta función recibirá como primer argumento el número al que se debe calcular la raíz cuadrada, su segundo argumento será el valor del error que se desea para el criterio de decisión. Es importante resaltar el hecho que muchos de los resultados que produce este algoritmo son aproximaciones de las respuestas correctas.

> (raíz 1 0.0001)

0.9999694824218...

> (raíz 2 0.0001)

1.4141845703125...

> (raíz 3 0.0001)

1.7320404052734...

> (raíz 4 0.0001)

?

> (raíz 9 0.0001)

2.9999885559082...

A continuación se presenta el código para el problema anterior. Se han utilizado funciones auxiliares para simplificar la especificación del problema.

```
(define (cálculo-de-Raíces por Bisección)
  (lambda (y err)
    (cond ((<= y 0) "Error")
          ((< err 0) "Error")
          ((<= err 0.0001) y)
          (else
            (let ((approx (raíz-aux y (/ y 2) err)))
              (if (buena-aproximación? approx err)
                  approx
                  (cálculo-de-Raíces approx err))))))

(define (raíz-aux inf sup approx err)
  (cond ((<= (abs (- approx (/ inf sup 2)))) err)
        ((<= (abs (- approx (/ inf sup 2)))) (* err 0.0001))
        (else
          (let ((mid (/ (+ inf sup) 2)))
            (if (<= (abs (- approx mid))) mid
                (if (<= (abs (- approx (/ inf mid 2)))) (/ inf mid)
                    (if (<= (abs (- approx (/ mid sup 2)))) (/ mid sup)
                        (cálculo-de-Raíces (/ mid sup 2) err))))))))
```

```

(raíz-aux      inf
           aprox
           (mejor-aproximación inf aprox)
           y
           err))
( (< (* aprox aprox) y)
  (raíz-aux aprox
  sup
  (mejor-aproximación aprox sup)
  y
  err)))))

;; Determina si Aprox está lo suficientemente
;; cerca de la raíz de y
;;
(define (buena-aproximación? aprox y err)
  (<= (abs (- y (* aprox aprox)))
       err))

;; Construye una nueva aproximación
;; con base en dos límites
;;
(define (mejor-aproximación inf sup)
  (/ (+ inf sup) 2))

```

Cálculo de raíces por Newton-Raphson

Una forma muy eficiente para calcular la raíz cuadrada de un número es la fórmula de Newton-Raphson. Se trata de una fórmula recurrente que genera mejores aproximaciones que el método de bisección.

El algoritmo inicia con una solución inicial de 1. Si se satisface el criterio de decisión se presenta la solución, en caso contrario se debe generar una mejor aproximación hasta que se logre satisfacer el criterio de selección. A continuación se presenta la fórmula para generar una mejor aproximación, en esta fórmula Y representa el número al que se le desea calcular la raíz, y Xn representa el conjunto de aproximaciones sucesivas.

$$X_0 = 1$$

$$X(n+1) = \left(\frac{1}{2} \right) * \left(X(n) + \frac{y}{X(n)} \right)$$

A continuación se presenta un ejemplo del método. En esta caso se desea encontrar la raíz cuadrada de 2 cuya raíz es de 1.4142... El criterio de decisión que se utilizará está dado por:

$$\left| 2 - Aprox^2 \right| \leq \text{Error}$$

$$\left| 2 - Aprox^2 \right| \leq 0.0001$$

Se tiene que $Y = 2$ y se establece un solución inicial de 1, es decir $X_0 = 1$

Al utilizar el criterio de decisión se tiene que :

$$\left| 2 - 1^2 \right| = \left| 2 - 1 \right| = 1$$

y

$$1 > 0.0001$$

Por lo que se debe generar una nueva aproximación:

$$X_1 = \left(\frac{1}{2} \right) \left(X_0 + \frac{y}{X_0} \right)$$

$$X_1 = \left(\frac{1}{2} \right) \left(1 + \frac{2}{1} \right)$$

$$X_1 = 1.5$$

Al utilizar el criterio de decisión con $X1 = 1.5$ se tiene que:

$$\left| 2 - 1.5^2 \right| = \left| 2 - 2.25 \right| = 0.25$$

y

$$0.25 > 0.0001$$

Se genera una nueva aproximación:

$$X2 = \left(\frac{1}{2} \right) * \left(X1 + \frac{y}{X1} \right)$$

$$X2 = \left(\frac{1}{2} \right) * \left(1.5 + \frac{2}{1.5} \right)$$

$$X2 = 1.4166\dots$$

Al utilizar el criterio de decisión con $X2 = 1.4166\dots$ se tiene que:

$$\left| 2 - 1.4166 \right| = \left| 2 - 1.3147 \right| = 0.0068$$

y

$$0.0068 > 0.0001$$

Se genera una nueva aproximación:

$$X3 = \left(\frac{1}{2} \right) * \left(X2 + \frac{y}{X2} \right)$$

$$X3 = \left(\frac{1}{2} \right) * \left(1.4166\dots + \frac{2}{1.4166\dots} \right)$$

$$X_3 = 1.4142\dots$$

Al utilizar el criterio de decisión con $X_3 = 1.4142\dots$

se tiene que:

$$\left| 2 - 1.4142^2 \right| = \left| 2 - 1.99996\dots \right| = 0.00004$$

$$0.00004 <= 0.0001$$

Al satisfacerse el criterio de decisión se establece que $X_3 = 1.4142\dots$ es la solución del problema propuesto.

Se construirá una función llamada raíz, que utiliza el mecanismo de Newton-Raphson. Esta función recibirá como primer argumento el número al que se le debe calcular la raíz cuadrada, su segundo argumento será el valor del error que se desea para el criterio de decisión. Se espera que el comportamiento de esta función sea similar a:

» (raíz 1 0.0001)

|

» (raíz 2 0.0001)

1.41421568627451...

» (raíz 3 0.0001)

1.73205081001473...

» (raíz 4 0.0001)

2.00000009292229...

» (raíz 9 0.0001)

3.00000000139698...

A continuación se presenta el código de dicha función:

- » Cálculo de Raíces por Newton-Raphson
- » Calcula la raíz cuadrada de Y
- » con un margen de error dado por Err
- » Precondiciones
- » y : real, y > 0

```

;; err: real, err > 0
;;
(define (raíz y err)
  (raíz-aux 1 y err))

(define (raíz-aux aprox y err)
  (cond ( (buena-aproximación? aprox y err)
          aprox)
        ( else
          (raíz-aux (mejor-aproximación aprox y)
                     y
                     err)))))

;; Determina si Aprox está lo suficientemente
;; cerca de la raíz de Y
;;
(define (buena-aproximación? aprox y err)
  (<= (abs (- y (* aprox aprox) ))
       err))

;; Construye una nueva aproximación
;; con base en dos límites
;;
(define (mejor-aproximación aprox y)
  (* (/ 1 2) (+ aprox
                  (/ y aprox))))
```

Estudio de la Raíz Cuadrada

En las secciones anteriores se han presentado dos algoritmos para realizar el cálculo de la raíz cuadrada de un número. Es natural preguntarse cuál de los dos algoritmos anteriores realiza este cálculo de una manera más rápida, o equivalentemente de forma más eficiente.

En las ciencias de la computación el estudio de tiempos de algoritmos se conoce con el nombre de análisis de algoritmos. Este rama de la computación utiliza formalismos matemáticos para estudiar la eficiencia de una solución propuesta. En esta sección será suficiente hacer un estudio experimental probando los dos algoritmos propuestos con distintos valores.

Tanto el método de Bisección como el método de Newton-Raphson dependen de una medida del error para detenerse. Entre menor sea el error que se solicita mayor será el tiempo que requiere el algoritmo para terminar. De forma equivalente entre mayor sea la exactitud que se necesite, mayor será el tiempo de procesamiento.

Para el caso en discusión el número de iteraciones que realiza cada algoritmo es una medida proporcional al tiempo que tardan. A continuación se presenta una tabla donde se calcula la raíz cuadrada de 2 contra diferentes medidas de error, la tabla indica el número de iteraciones que se deben efectuar para realizar el cálculo solicitado de acuerdo a un error establecido.

Cuadro N° 1

Número de Iteraciones de acuerdo a un error dado para cada método de la raíz cuadrada.

	Error 0.00001	Error 0.0000	Error 0.0000
Algoritmo		00001	0000
Bisección	15 iteraciones	29 iteraciones	42 iteraciones
Newton- Raphson	3 iteraciones	4 iteraciones	5 iteraciones



Cuadro N° 2

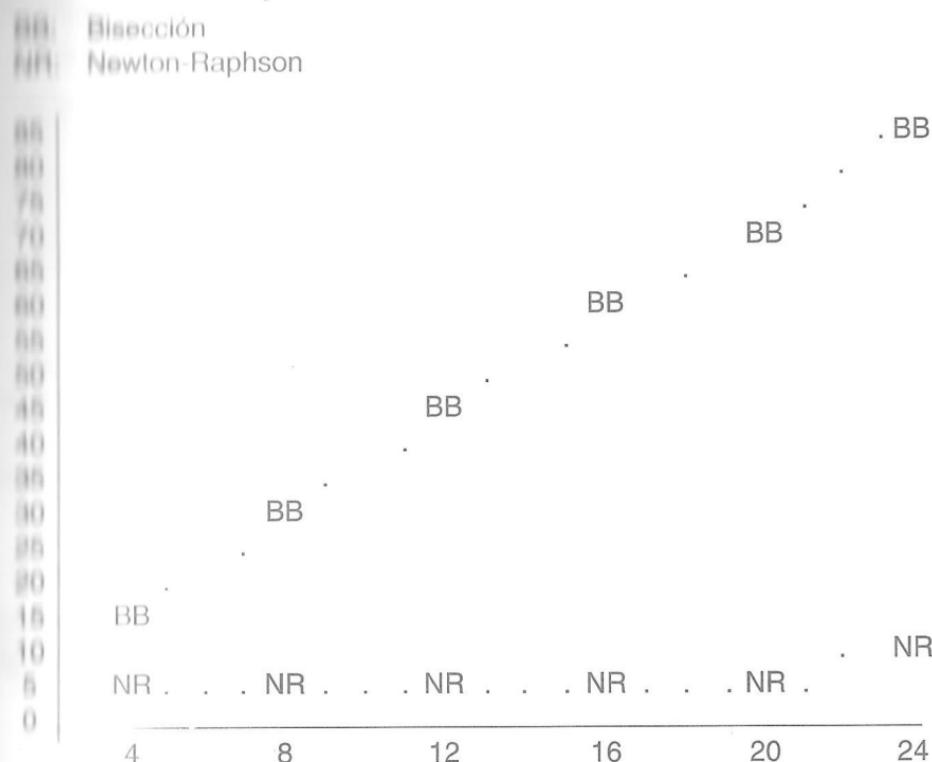
Número de Iteraciones de acuerdo a un error dado para cada método de la raíz cuadrada.

	Error	Error	Error
Algoritmo	0.0000	0.0000	0.0000
	0000	0000	0000
	0000	0000	0000
	00001	0000	0000
	00001	0000	00001
Bisección	56 iteraciones	70 iteraciones	84 iteraciones
Newton- Raphson	5 iteraciones	5 iteraciones	6 iteraciones

De igual forma, la tabla anterior se puede representar gráficamente. Observe como ambos algoritmos tienen un comportamiento lineal, sin embargo el número de iteraciones del método de bisección aumenta mucho más rápido que el número de iteraciones del método de Newton-Raphson.

Gráfico N° 1

Comparación de Bisección y Newton Raphson con base en el error y el número de iteraciones realizadas



Resumen

- *Definir* y *cond* son los principales mecanismos para la construcción y abstracción de funciones.
- La *recursión* es un elemento importante de control.
- Se pueden modelar funciones y expresiones matemáticas con facilidad utilizando los elementos anteriores.
- Para facilitar el proceso de programación un problema se debe dividir en subproblemas más pequeños y combinarlos.
- Cuando se construye un programa es importante estudiar el tiempo que tarda en ejecutarse. Esta es el área de estudio del análisis de algoritmos.

Ejercicios

Ejercicio N° 1.

Escriba utilizando la notación de Scheme, la representación de las siguientes fórmulas.

$$x^2 + y^2$$

$$(x + y)^2$$

$$\frac{2 * x * y}{w + z}$$

$$x^{n+1}$$

$$\frac{x}{x * y * z}$$

$$x^2 + 2*x*y + y^2$$

$$\frac{1}{x^2 + 2*x*y + y^2}$$

$$\frac{x^2 + 2*x*y + y^2}{x^4 + 2*x^2}$$

$$\frac{\left(\begin{array}{c} x^2 + 2*x*y + y^2 \end{array} \right)}{\left(\begin{array}{c} x^3 + 2*x*y + y^3 \end{array} \right)}$$

Ejercicio N° 2

Escriba una función para calcular la raíz más positiva reales de una ecuación cuadrática de la forma:

$$a \cdot x^2 + b \cdot x + c$$

Recuerde que dicha raíz se pueden encontrar mediante la fórmula:

$$r = \frac{b + \sqrt{b^2 - 4ac}}{2a}$$

La función deberá llamarse (equación a b c) y produce el valor de r como resultado.

Ejercicio N° 3

Construya una función que calcule la potencia de x elevado a la n, donde tanto x como n pueden ser cualquier número real. Para ello utilice las funciones del lenguaje de programación: (exp num) y (log num). A continuación se muestran algunos ejemplos de estas funciones:

- > (exp 0)
1
- > (exp 1)
2.71828182845905
- > (exp 2)
7.38905609893065
- > (log 0.01)
-4.60517018598809
- > (log 0.5)
0.693147180559945
- > (log 1)
0

```
> (log 2)
0.693147180559945
```

```
> (log 3)
1.09861228866811
```

Para programar la función (*eleva x n*) utilice la siguiente fórmula. Si n es un número entero positivo deberá utilizar la fórmula de multiplicaciones sucesivas que se mostró en el capítulo:

$$x^0 = 1$$

$$x^n = x * x^{(n - 1)} ; \text{ n entero y } n > 0$$

Si n es un entero negativo deberá utilizar esa misma fórmula e invertir el valor resultante pues:

$$x^{-n} = \frac{1}{x^n}$$

Finalmente si n es un número real utilice la siguiente fórmula:

$$x^n = \exp(n * \log(x))$$

Ejercicio N° 4

Se tiene una cuenta de ahorros en la cual se deposita un capital c al inicio de un año. Por cada año que transcurra se recibe un interés i sobre el capital depositado. Es decir al terminar el año se cuenta con el capital inicial más el interés ganado. Construya una función llamada (*int cap i n*) que calcula el monto que se recibe al depositar un capital a un interés dado durante un cierto número de años. Debe producir los resultados de acuerdo a la siguiente tabla:

Capital	Interés	Años	Resultado
2000	0.10	0	2000
2000	0.10	1	2200
2000	0.10	2	2420
2000	0.10	3	2662

La siguiente tabla indica que se deben producir los siguientes resultados:

$\text{elevar } (0) \quad (0)$

(0)

$\text{elevar } (0) \quad (0) \quad (0) \quad (0)$

(0)

$\text{elevar } (0) \quad (0) \quad (0) \quad (0) \quad (0)$

(0)

$\text{elevar } (0) \quad (0) \quad (0) \quad (0) \quad (0) \quad (0)$

(0)

$\text{elevar } (0) \quad (0) \quad (0) \quad (0) \quad (0) \quad (0) \quad (0)$

(0)

$\text{elevar } (0) \quad (0)$

(0)

$\text{elevar } (0) \quad (0)$

(0)

$\text{elevar } (0) \quad (0)$

(0)

$\text{elevar } (0) \quad (0)$

(0)

$\text{elevar } (0) \quad (0)$

(0)

$\text{elevar } (0) \quad (0)$

(0)

Ejercicio N° 5

A continuación se presentan un conjunto de fórmulas matemáticas. Cada una de estas fórmulas recibe un valor n entero, con $n > 0$, como el parámetro de entrada al programa y debe generar las operaciones respectivas. Construya una función en Scheme que calcule las fórmulas siguientes.

$$1 + 4 + 6 + \dots + 2n$$

$$4 + 8 + 12 + \dots + 4n$$

$$9 + 10 + 15 + \dots + 5n$$

$$1 + 3 + 5 + \dots + (2n-1)$$

$$1 + 5 + 9 + \dots + (4n-3)$$

$$2 + 5 + 8 + \dots + (3n-1)$$

En la situación anterior se deposita un capital c al inicio de un año. Si anualmente se recibe un interés i sobre el capital inicial, el año se cuenta con el capital inicial más el interés correspondiente. La función llamada (*int cap in*) que calcula el monto total a un interés dado durante un cierto número de años de acuerdo a la siguiente tabla:

Ejercicio N° 6

A continuación se presentan un conjunto de fórmulas matemáticas. Cada una de estas fórmulas recibe un valor n entero, con $n > 0$, como el parámetro de entrada al programa y debe generar las operaciones respectivas. Construya una función en Scheme que calcule las fórmulas siguientes.

$$1^2 + 3^2 + 5^2 + \dots + (2n-1)^2$$

$$1^3 + 2^3 + 3^3 + \dots + n^3$$

$$2^1 + 2^2 + 2^3 + \dots + 2^n$$

Años	Resultado
0	2000
1	2200
2	2420
3	2662

$$1*2 + 2*3 + 3*4 + \dots + n(n+1)$$

$$\frac{1}{1*2} + \frac{1}{2*3} + \frac{1}{3*4} + \dots + \frac{1}{n(n+1)}$$

$$\frac{1}{1*2*3} + \frac{1}{2*3*4} + \frac{1}{3*4*5} + \dots + \frac{1}{n(n+1)(n+2)}$$

Ejercicio N° 7

Se tiene la siguiente fórmula matemática. Construya una función que realice este cálculo. ¿Cuántos y cuáles parámetros y debe recibir la función?

$$1 + a + a^2 + a^3 + \dots + a^{n-1}$$

Ejercicio N° 8

A continuación se presentan un conjunto de sumatorias infinitas. Construya un programa en Scheme para descubrir el valor hacia el cual convergen estas sumatorias.

Para ello, debe programar la sumatoria y utilizar un número de n que tienda hacia infinito, es decir, un valor de n lo suficientemente grande para que el resultado de la sumatoria quede cerca del resultado. Cada resultado le indica hacia donde se cree que converge dicha sumatoria.

$$\sum_{i=0}^n \frac{1}{(i+1)(i+2)}$$

$$\sum_{i=0}^n \frac{1}{(i+1)(i+2)(i+3)}$$

$$\frac{1}{1+3} + \frac{1}{5+7} + \frac{1}{9+11}$$

Ejercicio N° 10

Escriba un programa en Scheme para calcular el cociente y el residuo al dividir n utilizando una fórmula que no utilice la función mod. A continuación se presenta la función quotient que representa la división

Si n es par:

$$x^n = quotient(n, 2) * quotient(n, 2)$$

$$\sum_{i=0}^n \frac{i}{2^i}$$

$$\sum_{i=0}^n \frac{i^2}{2^i}$$

$$\sum_{i=0}^n \frac{i^3}{2^i}$$

$$\sum_{i=0}^n \frac{x^i}{i!} = e^x$$

Ejercicio N° 9

A continuación se presenta la sumatoria de Leibniz que converge lentamente hacia un octavo de pi. Utilizando esta sumatoria, construya una función en Scheme que aproxime el valor de pi.

$$\frac{1}{1 * 3} + \frac{1}{5 * 7} + \frac{1}{9 * 11} + \dots = \frac{\pi}{8}$$

Ejercicio N° 10

Escriba un programa en Scheme para elevar un valor de x a una potencia entera positiva n utilizando una fórmula que calcula el resultado obteniendo valores intermedios. A continuación se presenta la fórmula matemática. Recuerde que la función quotient representa la división entera.

Si n es par:

$$x^n = x^{quotient(n,2)} * x^{quotient(n,2)}$$

Si n es impar:

$$x^n = x * x^{quotient(n,2)} * x^{quotient(n,2)}$$

Ejercicio N° 11

La sucesión de Fibonacci se define como una relación de recurrencia lineal de grado dos de la forma:

$$fib(0) = 1$$

$$fib(1) = 1$$

$$fib(n) = fib(n-1) + fib(n-2)$$

A toda relación lineal de grado dos se le puede asociar un polinomio de igual grado de la forma:

$$fib(n) = fib(n-1) + fib(n-2)$$

$$x^2 = 1 * x^1 + 1 * x^0$$

La cual se puede reducir a:

$$x^2 = x + 1$$

$$x^2 - x - 1 = 0$$

Al resolver esta ecuación se obtendrán dos raíces s1 y s2.

$$s1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$s1 = \frac{-1 - \sqrt{1^2 - 4(1)(-1)}}{2(1)}$$

$$s1 = \frac{1 - \sqrt{5}}{2}$$

y de forma análoga, se obtiene que

$$s2 = \frac{1 + \sqrt{5}}{2}$$



Finalmente se conoce que :

$$f(n) = u * s1^n + v * s2^n$$

De donde al utilizar las condiciones iniciales se obtiene

$$f(1) = u * s1^1 + v * s2^1$$

$$1 = u * s1^1 + v * s2^1$$

$$f(2) = u * s1^2 + v * s2^2$$

$$1 = u * s1^2 + v * s2^2$$

Que al sustituir el valor de las raíces $s1$ y $s2$ se convierte en:

$$f(1) = u + \left(\frac{1 - \sqrt{5}}{2} \right)^1 + v + \left(\frac{1 + \sqrt{5}}{2} \right)^1$$

$$f(2) = u + \left(\frac{1 - \sqrt{5}}{2} \right)^2 + v + \left(\frac{1 + \sqrt{5}}{2} \right)^2$$

Observe que esto genera un sistema de ecuaciones con dos incógnitas u y v y dos ecuaciones. Por lo que es posible encontrar los valores de u y v .

- Resuelva el sistema de ecuaciones para u y v .
- Escriba un programa en Scheme que genere el n -ésimo número de Fibonacci utilizando el método siguiente. ¿Nota alguna diferencia entre los resultados producidos por este método y el método convencional?

$$f(n) = \frac{-1}{\sqrt{5}} + \left(\frac{1 - \sqrt{5}}{2} \right)^n + \frac{1}{\sqrt{5}} + \left(\frac{1 + \sqrt{5}}{2} \right)^n$$

Ejercicio N° 12

La siguiente sucesión numérica recibe únicamente valores de n enteros mayores o iguales a cero. Programe la esta sucesión numérica.

$$f(0) = 0$$

$$f(1) = 1$$

$$f(n+2) = 2 f(n) + f(n+1)$$

Sugerencia, primero convierta la sucesión a una forma de recurrencia que utilice el valor n de forma directa y luego programe la función de Scheme. Los primeros cinco valores de esta sucesión son 0,1,1,3,5,11.

Ejercicio N° 13

Construya una función de nombre (truncar n) que dado un valor n real le quite los decimales a dicho valor. Para la construcción de esta función NO puede utilizar la función de Scheme (truncate n) ni (round n). Por ejemplo:

```
> (truncar 2.33)
```

```
2
```

```
> (truncar 12.77)
```

```
12
```

```
> (truncar 31.97)
```

```
31
```

Ejercicio N° 14

Construya una función de nombre (redondear n) que dado un valor n realice un redondeo del número. Para la construcción de esta función NO puede utilizar la función (round n) ni (truncate n). Por ejemplo:

```
> (round 6.33)
```

```
6.0
```

```
> (round 6.5)
```

```
6.0
```

```
> (round 6.66)
```

```
7.0
```

Ejercicio N° 15

Escriba un programa para determinar si un número n positivo es primo o no. Recuerde que n es primo si es divisible únicamente por el valor de 1 y n.

En este nuevo algoritmo se tratará de probar si n es divisible por 2,3,...,k donde k debe ser impar y además $2*k < n$.

Si es divisible por alguno de estos números no es primo, si no es divisible por ninguno de ellos si es primo. La función debe tener el siguiente comportamiento.

```
> (primo? 1)
```

```
#t
```

```
> (primo? 2)
```

```
#t
```

```
> (primo? 5)
```

```
#t
```

```
> (primo? 9)
```

```
#f
```

```
> (primo? 11)
```

```
#t
```

Ejercicio N° 16

El abogado y miembro del parlamento de Toulouse, Pierre de Fermat (1601-1665) se dedicó a la matemática en sus tiempos libres. Uno de sus tantos enunciados fue tratar de construir una fórmula para la generación de números primos. Esta fórmula es la siguiente:

```
> (+ 1 (expt 2 (expt 2 n)))
```

```
;;para cualquier n entero, n=0,1,2,3,...
```

Desafortunadamente, aproximadamente 100 años después, Leonardo Euler (1707-17983) demostró que esta fórmula no es correcta.

- Construya un programa que dado un valor de n construya un número de Fermat.
- Construya un programa que dado un valor de n evalúe si el número generado es primo o no.

Ejercicio N° 17

Escriba un programa para determinar el número de primos que existe entre dos números enteros a y b. Suponga que 1 es un número primo. El resultado debe ser el siguiente:

```
> (num-primos 1 3)  
3  
> (num-primos 1 7)  
5  
> (num-primos 1 15)  
7
```

Ejercicio N° 18

Construya una función de nombre (primo-cercano num). Esta función encuentra el número primo p que cumple las siguientes condiciones. p es el número más cercano a num tal que p es primo y p <= num. Por ejemplo:

```
> (primo-cercano 10)  
7  
> (primo-cercano 11)  
11  
> (primo-cercano 20)  
19  
> (primo-cercano 65)  
61
```

Ejercicio N° 19

Construya una función de nombre (supp num), esta función indica cuántos números primos, iniciando del menor al mayor, deben sumarse tal que la suma de los mismos sea mayor o igual que num. Por ejemplo:

```
> (supp 1)
1
;; se deben sumar el número 1.

>(supp 10)
4
;; se deben sumar los números 1,2,3,5.

> (supp 20)
6
;; se deben sumar los números 1,2,3,5,7,11.
```

Ejercicio N° 20

Construya una función de nombre (fib-cercano num). Esta función encuentra un número p que cumple las siguientes condiciones. fib(p) es el numero más cercano a num tal que fib(p) <= num. Por ejemplo:

```
> (fib-cercano 10)
6
;; fib(6)=8 y fib(7)=13

> (fib-cercano 20)
7
;; fib(7)=13 y fib(8)=21

> (fib-cercano 65)
10
;; fib(10)=55 y fib(11)=89
```

Ejercicio N° 21

Escriba un programa que recibe un número entero positivo e indica el número de dígitos que lo compone.

Sugerencia: utilizar las funciones de (quotient n 10) y (remainder n 10). A continuación se presentan algunos ejemplos:

```
> (dígitos 568)  
3  
> (dígitos 12447)  
5  
> (dígitos 1298658)  
7
```

Ejercicio N° 22

Escriba un programa que recibe un número real positivo e indica el número de dígitos que lo compone. Observe que el número puede o no tener decimales.

Sugerencia: utilizar las funciones de (quotient n 10) y (remainder n 10). A continuación se presentan algunos ejemplos:

```
> (dígitos 568)  
3  
> (dígitos 568.75)  
5  
> (dígitos 3564.9876)  
8
```

Ejercicio N° 23

Escriba un programa que determine la suma de los dígitos de un número n, donde n es un entero positivo mayor estricto que cero.

Sugerencia: utilizar las funciones de (quotient n 10) y (remainder n 10). A continuación se presentan algunos ejemplos:

```
> (sumd 124)  
7  
> (sumd 1248)  
15  
> (sumd 10000)  
1
```

Ejercicio N° 24

Utilice el siguiente algoritmo para determinar si un número es divisible por 3. Sumar los dígitos del número de manera sucesiva hasta que la suma sea estrictamente menor que 10. Si la suma es 3,6 o 9 entonces el número es divisible por 3, en caso contrario no lo es.

Por ejemplo:

Para el número 12456711,
su suma es $1 + 2 + 4 + 5 + 6 + 7 + 1 + 1 = 18$

Luego se toma 18
su suma es $1 + 8 = 9$

Como es 3,6,9 es divisible por tres.

En general el algoritmo debe funcionar así:

```
> (div3 12)  
#t  
> (div3 111)  
#t  
> (div3 113)  
#f  
> (div3 12456)  
#t
```

Ejercicio N° 25

Escriba un programa que cuente el número de veces que aparece un dígito dentro de un número cualquiera, mayor estricto que cero. Por ejemplo:

```
> (contar 2 1223)  
2  
> (contar 1 1011)  
3  
> (contar 7 70771007)  
5
```

Ejercicio N° 26

Escriba un programa que invierta un número. La función debe obtener cada uno de los valores y construir un nuevo número que sea inverso del original. Por ejemplo:

```
> (invertir 700)  
007  
;;de manera equivalente puede responder 7  
> (invertir 766)  
667  
> (invertir 1542)  
2451
```

Ejercicio N° 27

Escriba un programa que sume un número específico a todos los dígitos de otro número. Si la suma es mayor que 10 se debe tomar el dígito menos significativo de la suma. Por ejemplo:

```
> (sumardd 7 1221)  
8998  
> (sumardd 7 1024 )  
8791  
> (sumardd 11 1024)  
2135
```

Ejercicio N° 28

Dos números se llaman cercanos si la suma de sus divisores es da el mismo resultado.

Todos los números primos son cercanos, pues su único divisor es 1.

De esta forma 13 y 7 son cercanos:

El único divisor de 7 es 1

El único divisor de 13 es 1

Los números 18 y 51 son cercanos, pues:

Los divisores de 18 son (1 2 3 6 9) cuya suma es 21

Los divisores de 51 son (1 3 17) cuya suma es 21

Los números 98 y 175 son cercanos, pues:

Los divisores de 98 son (1 2 7 14 49) cuya suma es 73.

Los divisores de 175 son (1 5 7 25 35) cuya suma es 73.

Los números 220 y 562 son cercanos, pues:

Los divisores de 220 son (1 2 4 5 10 11 20 22 44 55 110) y suman 284.

Los divisores de 562 son (1 2 281) y suman 284.

Programe una función de nombre (cercanos num1 num2) que determine si num1 y num2 son cercanos. Los parámetros de la función serán enteros positivos, es decir, enteros mayores estrictos que cero.

Ejercicio N° 29

Escriba un programa en Scheme que calcule la suma de:

$$G(1) + G(2) + G(3) + \dots + G(n)$$

Si se sabe que G(K) está definido por:

$$G(k) = \frac{1}{1} + \frac{1}{2} + \dots + \frac{1}{k}$$

Ejercicio N° 30

A continuación se presenta el método de Newton-Raphson para el cálculo de raíces cúbicas por aproximaciones sucesivas.

Si se desea calcular la raíz cúbica de un número real, denominado Y, se utiliza una primera aproximación denominada X(0) y se van construyendo una sucesión de valores que converge hacia el valor cúbico de Y.

El método se describe mediante la siguiente relación de recurrencia:

$$X(0) = 1$$

$$X(n+1) = \frac{1}{3} * \left(2*X(n) + \frac{Y}{X(n) * X(n)} \right)$$

Construya una función que utilice las fórmulas descritas para calcular el valor de la raíz cúbica de Y.

Ejercicio N° 31

Programación Avanzada: Cambio de Base Numérica.

Un número decimal por ejemplo 1274 toma su valor de una base 10, por ejemplo:

$$\begin{array}{r} 4 * 1 \\ + 7 * 10 \\ + 2 * 100 \\ + 1 * 1000 \\ = 1274 \end{array}$$

En este caso se dice que 1274 se encuentra expresado en base decimal. Los computadores sin embargo utilizan una base binaria, por ejemplo el número binario 1010 representa un 10 decimal ya que:

$$\begin{array}{r} 0 * 1 \\ + 1 * 2 \\ + 0 * 4 \\ + 1 * 8 \\ = 10 \end{array}$$

Para convertir un número de Base 10 (Base Decimal) a Base 2 (Base Binaria) se deben realizar un conjunto de divisiones sucesivas entre 2. A continuación se esboza este proceso:

$$\begin{aligned} \text{quotient}(10,2) &= 5 \quad \text{remainder}(10,2) = 0 \\ \text{quotient}(5,2) &= 2 \quad \text{remainder}(5,2) = 1 \\ \text{quotient}(2,2) &= 1 \quad \text{remainder}(2,2) = 0 \\ \text{quotient}(2,1) &= 0 \quad \text{remainder}(2,1) = 1 \end{aligned}$$

Observe como al unir todos los valores de la función remainder se obtiene el valor de 1010.

- a) Con base en la descripción anterior construya una función que convierta cualquier número base 2 a base 10. Por ejemplo:

```
> (base10 '10)
2
```

```
> (base10 '1010)
```

```
10
```

```
> (base10 '1110)
```

```
14
```

- b) Construya una función que convierta cualquier número base 10 a base 2. Por ejemplo:

```
> (base2 '2)
```

```
10
```

```
> (base2 '10)
```

```
1010
```

```
> (base2 '14)
```

```
1110
```

Ejercicio N° 32. Programación Avanzada: Intersección de Funciones.

Sean $f_1(x)$ y $f_2(x)$ dos rectas definidas como:

$$f_1(x) = x * m_1 + b_1$$

$$f_2(x) = x * m_2 + b_2$$

Construya una función que se llame intersección que recibe los siguientes argumentos: (intersección m_1 b_1 m_2 b_2). Dicha función debe retornar el valor de `#f`, si las rectas no tienen intersección y debe devolver el valor de la intersección en caso contrario.

A continuación se presentan unos ejemplos de como se espera que se comporte dicha función:

```
;; Note que
;; f1(x) = 2 * x + 2
;; f2(x) = 2 * x + 20
;;
> (intersección 2 2 2 20)
#f
```

```
;; Note que
;; f1(x) = 2 * x + 2
;; f2(x) = -1 * x + 4
```



```

;;
> (intersección 2 2 -1 4)
0.666...
;; Note que
;; f1(x) = 6 * x + 4
;; f2(x) = -2 * x + 3
;;
> (intersección 6 4 -2 3)
-0.125

```

Ejercicio N° 33. Programación Avanzada: Más Intersección de Funciones.

Si se tienen cualesquiera dos puntos de una recta de la forma (x_1, y_1) y (x_2, y_2) se puede construir la función simbólica.

Sean (x_1, y_1) y (x_2, y_2) dos puntos de una misma recta.

Asimismo sabemos que toda recta se puede expresar por la relación:

$$y = m x + b$$

Donde m es la pendiente de la recta y b la intersección sobre el eje y .

Luego los valores de m y b se pueden construir con base en:

$$m = \frac{y^2 - y^1}{x_2 - x_1}$$

$$b = y - m x ; \text{ para cualquier valor de } (x, y)$$

Por ejemplo suponga que:

$$(x_1, y_1) = (0, 2)$$

$$(x_2, y_2) = (1, 4)$$

$$m = \frac{4 - 2}{1 - 0}$$

Usando el punto $(x_1, y_1) = (0, 2)$ para localizar:

$$b = 2 - 2(0) = 2$$

Por lo que podemos construir la función final:

$$\begin{aligned}y &= 2x + 2 \\f(x) &= 2x + 2\end{aligned}$$

Construya una función que se llame intersección-puntual, esta función recibe cuatro puntos $(x_{1a}, y_{1a}), (x_{1b}, y_{1b})$ y $(x_{2a}, y_{2a}), (x_{2b}, y_{2b})$.

La función

(intersección-puntual $x_{1a} \ y_{1a}$
 $x_{1b} \ y_{1b}$
 $x_{2a} \ y_{2a}$
 $x_{2b} \ y_{2b}$)

realiza el siguiente proceso:

Con (x_{1a}, y_{1a}) y (x_{1b}, y_{1b}) construye una función $f_1(x)$.

Con (x_{2a}, y_{2a}) y (x_{2b}, y_{2b}) construye la función $f_2(x)$.

Encuentra la intersección de $f_1(x)$ y $f_2(x)$.