UTP-ISC

Guía de estudio

Programación Funcional con Racket.

Luis Eduardo Muñoz Guerrero

Guía de estudio:

Programación Funcional con ((Racket))

Luis Eduardo Muñoz
Profesor titular
Universidad Tecnológica de Pereira

Universidad Tecnológica de Pereira
Facultad de Ingenierías
Ingeniería en sistemas y computación
Octubre de 2014

Primera Edición, Pereira – Risaralda, Octubre de 2014

Luis Eduardo Muñoz Guerrero. Autor

Jose Alejandro Cardona Valdes. Coautor. Diseño.

Edición:

//Dedicatorias, notas del autor

CONTENIDO

ÍNDICE

INTRODUCCIÓN

Este libro nació del interés por crear una guía completa que ayude a los estudiantes de primer semestre de la Universidad Tecnológica de Pereira. -dado que es muy poca la documentación completa que existe acerca del lenguaje de programación Racket en español-

Además, este libro pretende ser una guía para las personas que empiezan a descubrir el mundo de la programación, utilizando un lenguaje natural y poco técnico a la hora de explicar los temas expuestos y basándonos en un paradigma de programación fácil de entender como lo es el Funcional, con ejemplos y actividades que conlleven al desarrollo de unas buenas habilidades para programar y un conocimiento medio del lenguaje de programación Racket.

1

INTRODUCCIÓN A LA PROGRAMACIÓN

Es lógico pensar en que, para crear un programa de computadora hay que darle instrucciones a la misma y por consiguiente debemos saber cómo hacerlo, en otras palabras, hablar su mismo idioma. De entrada esto supone un problema dado que no es fácil hablar en el idioma de una maquina pues este solo contiene dos "palabras" que son: cero y uno. A este lenguaje se le llama: código máquina, y corresponde al más bajo nivel.

Dada la dificultad que supone el código máquina, los primeros científicos que trabajaban en el área de la computación empezaron a desarrollar métodos más sencillos de comunicación con la máquina, codificaban secuencias de ceros y unos haciéndolas corresponder a palabras propias del inglés y creando así un lenguaje de más nivel. A este lenguaje se le llamo: El ensamblador.

A medida que la complejidad de las tareas que realizaban las computadoras aumentaba, se hizo necesario disponer de un lenguaje aún más sencillo que el ensamblador para programar. Entonces, se crearon los lenguajes de alto nivel. Se crearon lenguajes más parecidos a los de los humanos. Mientras que una tarea tan trivial como sumar dos números podía necesitar un conjunto de instrucciones en lenguaje ensamblador, en un lenguaje de alto nivel bastará con solo una y muy similar a lo que usamos normalmente.

Sin embargo, sigue siendo necesario que ese lenguaje de alto nivel sea traducido a código máquina, es por eso que se crearon los "Compiladores" o "Interpretes", ellos son los encargados de esta tarea.

1.1 QUE ES LA PROGRAMACIÓN

La programación es el proceso de diseñar, escribir, probar, depurar y mantener el código de programas computacionales. El código fuente es escrito en un lenguaje de programación. El propósito de la programación es crear programas que exhiban un comportamiento deseado. El proceso de escribir código requiere frecuentemente conocimientos en varias áreas distintas, además del dominio del lenguaje a utilizar, algoritmos especializados y lógica formal.

1.2 LOS PARADIGMAS DE PROGRAMACIÓN

Un paradigma de programación es fundamentalmente un estilo de programación de computadoras, una manera de construir la estructura y los elementos que conforman un programa, este estilo es, generalmente, ampliamente aceptado y adoptado por una comunidad de programadores.

Se trata tambien de la forma en la que se analiza, afronta y resuelve un problema computable.

Muchos lenguajes de programación adoptan paradigmas precisos mientras que otros soportan múltiples paradigmas, esto quiere decir que, cada lenguaje se especifica para que su uso se dé de acuerdo a determinado paradigma(s), por ejemplo, C++ se enfocó en soportar el paradigma orientado a objetos pero sin dejar de lado el paradigma estructurado.

Entre los principales paradigmas de programación se encuentran: Funcional, Orientado a objetos, Imperativo, Restricciones, Lógico y declarativo.

Veamos una pequeña descripción acerca de los más usados:

Imperativos: Son aquellos lenguajes, que basan su funcionamiento en un conjunto de instrucciones secuenciales, dichas instrucciones van alterando en la memoria los estados de las variables involucradas en la solución del problema, es decir, se cambia progresivamente el estado del sistema, hasta alcanzar la solución del problema.

Algunos lenguajes que usan este paradigma son: Basic, C, PHP, Fortran, Java

Orientados a Objetos: Este paradigma, generalmente se mezcla con alguno de los otros modelos, sin embargo mantiene características propias, que lo diferencian claramente. Los programas de este tipo, se concentran en los objetos que van a manipular, y no en la lógica requerida para manipularlos. Se trata de modelar el mundo en base a objetos, estos se crean para ser reutilizables computacionalmente sin tener que preocuparse de su lógica o comportamiento, solo de la utilidad propia de este. Ejemplos de objetos pueden ser: estudiantes, coches, casas etc., cada uno de los cuales tendrá ciertas funciones (métodos) y ciertos valores que los identifican (propiedades), teniendo además, la facultad de comunicarse entre ellos a través del paso de mensajes.

Algunos lenguajes que usan este paradigma son: C++, Java, .Net, Python.

Declarativos: La programación declarativa se basa en el desarrollo del programa especificando o "declarando" un conjunto de condiciones, proposiciones, afirmaciones, restricciones, ecuaciones o transformaciones que describen el problema Este paradigma abarco otro tales como el lógico, algebraico y funcional.

Algunos lenguajes que hacen uso del paradigma declarativo son: Haskell, Lisp, Prolog, Curry

Funcional: La programación funcional (que trataremos a fondo en este texto) se basa en la utilización de la estructura de función matemática, la cual posee entrada y parámetros que son procesados para obtener una salida (solución.). Enfatiza la aplicación de funciones a diferencia de la programación imperativa que enfatiza en los cambios de estado. Este paradigma tiene su raíces en el cálculo lambda.

Recordar: Un lenguaje de programación puede hacer uso de más de un paradigma.

1.3 Conceptos fundamentales de la programación funcional

La programación funcional apareció como un paradigma independiente a principio de los años sesenta. Su creación es debida a las necesidades de los investigadores en el campo de la inteligencia artificial y en sus campos secundarios del cálculo simbólico, pruebas de teoremas, sistemas basados en reglas y procesamiento del lenguaje natural.

La característica principal de la programación funcional es que los cálculos se ven como una función -matemática- que hacen corresponder entradas y salidas. No hay noción de posición de memoria y por tanto, necesidad de una instrucción de asignación. Los bucles se modelan a través de la recursividad ya que no hay manera de incrementar o disminuir el valor de una variable. Como aspecto práctico casi todos los lenguajes funcionales soportan el concepto de variable, asignación y bucle. Los lenguajes de Programación más representativos de la Programación Funcional, son: Racket (antes Scheme) y Haskell. Otra característica a tener en cuenta es su tipado débil o nulo en ciertos casos.

Podemos analizar la programación funcional desde el concepto de Función matemática, veamos:

Toda función tiene una entrada, un cuerpo y una salida, lo que sale está relacionado de alguna manera con lo que entra.

Ejemplos:

Sumar, Multiplicar, Restar y Dividir son funciones muy simples, entran dos números y sale otro al que llamamos resultado.

La raíz cuadrada también es una función. Seno, Coseno, y Tangente, son funciones que se usan en la trigonometría.

Si realizamos cada una de las funciones enunciadas en los ejemplos anteriores, veremos cómo se cumple el concepto de función, pues todas tienen una entrada un cuerpo (aunque no lo conozcamos) y una salida, lo que sale está relacionado, de alguna manera, con lo que entra.

Así mismo sucede en la programación funcional, existen muchas funciones ya definidas y nosotros podremos definir mas, todas tienen una entrada y una salida, solo que les llamaremos de una manera distinta.

Continuando con las bases matemáticas, veamos paso a paso como se construye una función:

Lo primero que se debe hacer es darle un nombre, matemáticamente el nombre más común es ***"f"***, pero en realidad puede dársele cualquier nombre, algo como ***"madera"*** si se prefiere.

Luego debemos nombrar lo que va dentro de la función (parámetros), se pone entre paréntesis y va después del nombre de la función:

Así ***f(x)*** indica que la función se llama ***"f"*** y que ***"x"*** es un parámetro.

Una función debe devolver un resultado:

$f(x) = x^3$ esto nos indica que la función ***"f"*** toma a ***"x"*** y lo eleva al cubo.

Ahora tenemos que, si a la función ***f(x)*** le damos una entrada de "2" dará como resultado: 8.

$f(2) = 8$

Bajo estas mismas nociones programaremos funcionalmente, usaremos nuestras propias funciones dependiendo del problema que deseemos resolver, y aunque nuestras funciones tengan una notación diferente a la matemática, en esencia son lo mismo, ya que se rigen con los mismos principios. Podremos verlo a grandes rasgos en el siguiente capítulo.

1.4 Notación Preorden, Recorrido de árbol binario

Existen tres tipos de notación a la hora de escribir expresiones. Estas notaciones se refieren a la posición del operador con respecto a los operandos. Para cada tipo de notación existe una secuencia o algoritmo que nos indica como es dicha posición:

Inorden:

Recorrer el subárbol izquierdo en Inorden
Examinar la raíz
Recorrer el subárbol derecho en Inorden

Preorden:

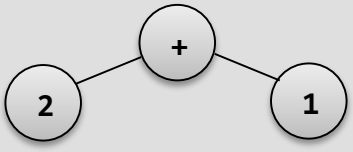
Examinar la raíz
Recorrer el subárbol izquierdo en Preorden

Recorrer el subárbol derecho en Preorden

Postorden:

Recorrer el subárbol izquierdo en Postorden
Recorrer el subárbol derecho en Postorden
Examinar la raíz

La notación de una expresión puede representarse con un Árbol Binario, si aplicamos el algoritmo correspondiente para recorrer dicho árbol obtendremos nuestra expresión en la notación deseada.

Inorden: **2 + 1** Preorden: **+ 2 1** Postorden: **2 1 +**		
Ejemplos:	(2 + (3 * 4)) = x	((2 + 3)* 4) = x
Preorden	= (+ (* 3 4) 2) x	= (* (+ 2 3) 4) x
Inorden	(2 + (3 * 4)) = x	((2 + 3) * 4) = x
Postorden	(2 (3 4 *) +) x =	(4 (2 3 +) *) x =

Como vemos, la notación a la cual estamos acostumbrados es Inorden, pero ahora debemos preocuparnos por aprender y entender la notación Preorden, ya que es la propia del paradigma Funcional y también de Racket. Veamos más ejemplos:

Inorden: `(x + 3) / (x - 7)`

Preorden: `/ (+ x 3) (- x 7)`

Inorden: `2+3+8+9+10+11+20+33+12`

Preorden: `(+(+(+(+(+(+(+(+ 2 3)8)9)10)11)20)33)12)`

Inorden: `(10+14)/20*124*8*(2-255)`

Preorden: `(*(*(*(/(+ 10 14)20)124)8)(- 2 255))`

Inorden: `1/2+1/3+1/5*(5+1)+20*(5+233)/2`

Preorden: `(/(+(*(+(/ 1 2)(/ 1 3)(/ 1 5))(+ 5 1))(* 20 (+ 5 233)))2)`

Inorden: `(3*9)+(2+2)/(5+3)`

Preorden: `(+(*3 9)(/(+ 2 2)(+ 5 3)))`

Nota: Es clave encerrar siempre nuestras operaciones con paréntesis, de esta forma evitaremos confusiones al momento de colocar nuestros operadores o de leer la expresión y a la hora de programar nos ahorrara errores en nuestro código.

1.5 ACTIVIDADES:

1. Pasar a notación Preorden:
    a. `-3`
    b. `8 = -3`
    c. `(2 * 3) + 5`
    d. `((5 + 2) * 3)`
    e. `(1 + 4) * (4 + 6)`
    f. `(3 * 3) / (8 * 2)`
    g. `5 + (6 / 2) + 3`
    h. `5 + (3 * 8) + 1`
    i. `((3 + 4) * 8) + 2`
    j. `(3 + ((8 - 2) – 4)) / 6`
    k. `(5 * (75 / 15)) + (4 * (4 - 1)) + (2 * (7 + 4))`
    l. `((15 / (8 - 3)) + (4 * (6 + 2))) * 2`
    m. `(8 + 3) * (40 - (7 * 4))`
2. Pasar de notación Preorden a notación Inorden
    a. `(* (+ 7 8) 3)`
    b. `(/ (+ 2 8) (* (+ 2 1) 4))`
    c. `(* (- (+ 8 -3) 1) 3)`
    d. `(- (+ A B) (+ A C))`
    e. `(/ (- (+ 1 1 ) 1) 1)`

2

INTRODUCCIÓN A RACKET

El lenguaje de programación Scheme apareció en la década del 1970 como un lenguaje de programación funcional y un dialecto de Lisp. Fue desarrollado por Guy L. Steele y Gerald Jay Sussman. Poco más de 20 años el proyecto cambio su nombre y paso a llamarse ***Lenguaje de Programación Racket,*** donde Scheme (R5RS) pasó a ser una extensión del lenguaje o un estándar más, soportado por el lenguaje.

2.1 CONCEPTOS FUNDAMENTALES DEL LENGUAJE RACKET

Racket, antes llamado PLT Scheme es un lenguaje de programación derivado de Scheme. Acepta la programación Funcional, Modular, Orientada a objetos y de procedimientos, lo que quiere decir que es Multi-paradigma. Su tipado puede ser Dinámico, Débil, Fuerte y Estático. Es multiplataforma, además es un lenguaje Interpretado.

Racket está bajo licencia GNU LGPL.

Aunque actualmente Racket sea un lenguaje Multi-paradigma a la hora de usarlo de manera funcional nos ofrece todas las capacidades y características de cualquier lenguaje puramente funcional. El uso de Racket en otros paradigmas de programación demanda un conocimiento más avanzado en el mismo y de la programación en general.

2.2 EL ENTORNO DE DESARROLLO INTEGRADO, DRRACKET

Es muy importante diferenciar entre una herramienta de programación, un lenguaje de programación y un entorno de desarrollo integrado (IDE).

Una herramienta de programación es ***un*** programa informático que usa un programador para crear, depurar y mantener programas, las herramientas de programación se enfocan en el desarrollo de programas en un lenguaje especifico, puede ser desde un editor de texto con resaltado de sintaxis e indentacion automática (notepad++, sublimetext) hasta un programa compilador (gcc, g++).

Como ya conocemos el concepto de leguaje de programación pasemos a ver que es un Entorno de desarrollo integrado:

Un entorno de desarrollo integrado es un programa informático, compuesto por un conjunto de herramientas de programación, puede dedicarse en exclusiva a un solo lenguaje de programación, o bien, a varios. Reúne un compilador o un intérprete, un editor de código, un depurador, un constructor de interfaz gráfica (GUI) entre otros.
Usamos "IDE" para referirnos a cualquier entorno de desarrollo integrado ya que este es el acrónimo de "Integrated Development Environment".
Algunos IDE conocidos son: DevC++, Eclipse, Microsoft Visual Studio, DrRacket, Netbeans, Geany, Ninja, QT Creator…
En conclusión, un IDE incorpora un conjunto de herramientas de programación específicas para uno o varios lenguajes de programación, construyendo así un programa informático completo para desarrollar otros programas.

2.2.1 Descarga, Instalación y Elementos de la ventana de DrRacket

DrRacket es el IDE con el que nos corresponde trabajar, para descargarlo debemos ir a la siguiente página web:

http://racket-lang.org/download/

Luego seleccionamos nuestra plataforma y damos clic al botón "Download" para iniciar la descarga del archivo.

Instalación en Windows:
Para instalar en Windows basta con hacer doble clic sobre el archivo ejecutable que descargamos y seguir los pasos en el programa de instalación.

Instalación en Linux:
Si nuestro sistema operativo es basado en Debían debemos descargar el archivo correspondiente a nuestra la plataforma: *"Linux i386 (Ubuntu ...)"*. Luego abrimos una terminal y nos ubicamos en la carpeta donde está el archivo descargado (generalmente la carpeta de descargas: /home/usuario/Descargas). Escribimos el comando "sh" seguido del nombre del archivo y su extensión.

```
$sh nombre_de_archivo.sh
```

Para finalizar respondemos las preguntas que nos haga el instalador acerca del lugar en donde colocara los archivos y los accesos directos al programa.

Luego de la instalación, para ejecutar DrRacket basta con escribir en una terminal el comando: `"drracket"` o buscar el lanzador en nuestra lista de programas.

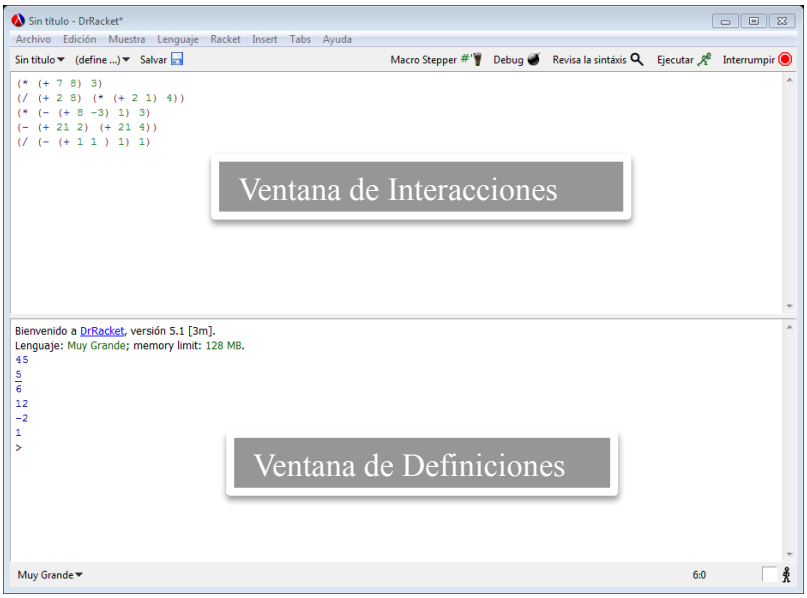
Para usuarios de Archlinux y derivadas existe un paquete en los repositorios AUR, con lo que bastara ejecutar el comando:

```
$yaourt –S racket
```

Para información adicional consulte la página web de Racket: http://racket-lang.org

La ventana de DrRacket

Es una interface muy sencilla, al iniciar nos encontraremos con dos elementos principales, la ventana de definiciones y la ventana de interacciones (donde se ejecutara nuestro código). Entre los demás elementos tenemos los menús, desde donde podemos cambiar el idioma del programa (*Help>Interactuar con DrRacet en Español*), el botón "Ejecutar" (para correr nuestro programa), Interrumpir (para interrumpir una ejecución en curso), Debug (para ejecutar el código instrucción a instrucción).



Para empezar a interactuar con DrRacket debemos escoger un lenguaje, esto lo hacemos dando clic en el botón de la esquina inferior izquierda de la ventana que dice: “Choose a language”, se despliega una lista, y escogemos la opción: “Seleccionar lenguaje…”.También podemos hacerlo desde el menú “Lenguaje > Seleccionar lenguaje”, o con el atajo del teclado “Ctrl + L”.

En la ventana emergente seleccionaremos: “Muy Grande”, clic en “Ok” y DrRacket estará listo para empezar a interpretar nuestro código.

Nota: En la mayoría de los ejemplos incluidos en este libro trabajaremos con el lenguaje “Muy Grande” dada la simplicidad de este. Eventualmente será necesario cambiar a “Estudiante avanzado” por lo que en los casos en los que sea necesario cambiar de lenguaje lo indicaremos al principio del código.

Como recomendación, para empezar a interactuar con el programa pueden ejecutar la solución de los ejercicios del punto uno de la página 14, y los ejercicios del punto dos.

2.3 La sintaxis básica

Veremos cómo se representan los datos en Racket, la notación del lenguaje y los tipos de datos.

En Racket todas las funciones deben ir dentro de paréntesis, corchetes o llaves. Al principio de cualquier función o entrada siempre encontraremos los siguientes caracteres:

( ) [ ] { } ; , “ ‘ #\

Es importante tener en cuenta que los operaciones aritméticas y todas las funciones, siempre deben ir encerradas entre paréntesis ().

Ejemplo: Si lo que queremos es realizar una suma:

Forma errónea: **+ 2 2**

Forma correcta: **(+ 2 2)**

En Racket tenemos muchos *tipos de datos*, trataremos de ver brevemente los que más vamos a usar:

Números:

Para operar con un número basta con teclearlo, indiferentemente de ser entero o flotante. A diferencia de la gran mayoría de lenguajes aquí no declararemos el tipo de dato.

Ejemplo:

```
7        -> 7
(+ 7 7)  -> 14
```

Como vemos, con los números y con la mayoría de datos en Racket, no es necesario declarar el tipo de dato que vamos a introducir.

Símbolos:

Los **Símbolos** se declaran con el carácter apostrofe:

Ejemplo:

```
'Esto-es-un-símbolo
'+23)( Racket…,,,
```

Los dos ejemplos anteriores, son considerados símbolos, sin importar la cantidad o los caracteres que este contenga y siempre y cuando no haya un espacio o un salto de línea, en tal caso los caracteres siguientes al espacio o salto de línea ya no serán considerados un símbolo.

Ejemplo:

```
'Este es -> Error: es: this variable is not defined
'Este.es -> Este.es
'2+2     -> 2+2
```

En los 3 ejemplos anteriores tenemos: El primero deja de ser un símbolo después del primer espacio y arroja como resultado un error que nos indica que la referencia "**es**" no está definida. El segundo y el tercero arrojan como resultado todos los caracteres considerados símbolos.

Cadenas:

Continuemos con las **Cadenas**. Para crear una cadena solo debemos encerrar los caracteres entre comillas dobles:

Ejemplo:

```
"Esta es una cadena"
"Esta 'También es una cadena ´+ 2 3 x [^^]"
```

Caracteres:

En Racket un **carácter** es de la siguiente forma:

Ejemplo:

```
#\J #\j #\3 #\+ ; Todos son caracteres
#\f
#\Hola     ->Read: bad character constant: #\Hola?
```

De los tres ejemplos anteriores solo los primeros dos son caracteres, el tercero en cambio, no lo es, ya que esta es una palabra completa y no un carácter, si quisiéramos escribir esta palabra como caracteres tendríamos que declarar cada uno de ellos, de la siguiente forma:

```
#\H #\o #\l #\a
```

Caracteres de agrupamiento:

Las funciones también podemos encerrarlas entre llaves y corchetes, no solo entre paréntesis. Aunque comúnmente veremos que se usan paréntesis para casi todo, es recomendable tener un orden de uso y alternar entre estos para aumentar la legibilidad del código.

Ejemplo:

```
(+ 5 6)         ->11
[+ 5 6]         ->11
{+ 5 6}         ->11
```

Obtendremos el mismo resultado en todos los casos, sin importar si usamos llaves, paréntesis o corchetes; Racket usa la notación Preorden es por eso que siempre debes poner el operador antes de los operandos (Ver 1.4).

Comentarios:

Para escribir un **Comentario** solo debemos antecederlo del carácter punto y coma, el comentario terminara en el siguiente salto de línea:

Ejemplo:

```
; Esto es un comentario
```

Booleanos:

Los **booleanos** son un tipo de dato que indica Falso o Verdadero, son el resultado de las sentencias de comparación, como lo sería 2 > 3, a lo que Racket arrojaría *#f* que es su forma de representar falso. Verdadero se representa *#t*.

Esta tabla contiene en resumen los tipos de datos que se manejan en Racket:

TIPO	EJEMPLO
Booleano	`#t, #f`
Entero	`1, -2, 3, 97, 0`
Racional	`1/4,  36/8`
Real	`3.1532,  1.2e+4`
Complejo	`0+i 15.9405,  2+4i`
Carácter	`#\c, #\a, #\i, #\3, #\1, #\space, #\tab, #\newline`
Símbolo	`‘var ‘xyz ‘programación`
Cadena	`“¡Hola- Mundo!”`
Lista	`‘(1 2 3 4 5 6)`
Vector	`#(1 2 3 4 5 6)`
Estructura	`(define-struct persona (nombre cc dirtel))`

Nota: La sintaxis de estos tipos de dato puede variar dependiendo del lenguaje que escogamos.

3

FUNCIONES

Retomaremos el concepto de función dado en el primer capítulo: toda función tiene una entrada y una salida, en la programación funcional esto no es diferente, toda operación que reciba datos y arroje otros es una función, de ahora en adelante siempre trabajaremos con funciones, bien sean las que ya están definidas en Racket o las que creemos nosotros.

Los objetivos de este capítulo son básicamente conocer las principales funciones primitivas para que hagamos un uso correcto y completo de ellas, además aprender a crear nuestras propias funciones y ejecutarlas (llamado a funciones).

3.1 FUNCIONES PRIMITIVAS

Llamamos Primitivas a las funciones que ya están definidas en el lenguaje, un ejemplo sencillo de estas sería la función suma "`+`" y la función "`display`", la primera recibe dos o más parámetros y devuelve un resultado que es la suma de estos, la segunda es una función usada para imprimir texto en pantalla.

Veamos las principales funciones primitivas que encontraremos en Racket, que tipo de datos reciben y arrojan.

Nota: La columna "Entrada -> Salida" contiene el tipo y el número de datos que recibe y que devuelve la función, Por ejemplo: la multiplicación "*****" tendrá la siguiente estructura: `(num num … -> num)` donde "num" es el tipo de dato: número, este aparece dos veces porque es la mínima cantidad de parámetros que puede operar y los puntos suspensivos significan que puede haber como máximo n parámetros, por último usaremos el símbolo -> para representar la salida que es de tipo: numero.

FUNCIONES QUE OPERAN CON VALORES NUMÉRICOS:

Función	Entrada -> Salida	Descripción
*	(num num … -> num)	Multiplicación
+	(num num … -> num)	Suma
-	(num num … -> num)	Resta
/	(num num … -> num)	Division
<	(real real … -> bool)	Compara el primer valor con los demás, si este es el menor devuelve: #t, sino #f
<=	(real real … -> bool)	Compara el primer valor con los demás, si este es menor o igual a los demás devuelve: #t, sino #f
>	(real real … -> bool)	Compara el primer valor con los siguientes, si este es el mayor obtendremos: #t, de lo contrario #f

>=	(real real … -> bool)	Compara el primer valor con los demás, si este es mayor o igual devuelve: #t, sino #f
abs	(real -> real)	Valor absoluto de un número real
acos	(num -> num)	arco-coseno
add1	(num -> num)	Aumenta en uno un numero
angle	(num -> real)	Calcula el Angulo de un #real
asin	(num -> num)	arco-seno
atan	(num -> num)	arco-tangente
ceiling	(real -> int)	
complex?	(any -> bool)	Evalúa si algo es o no un numero complejo
conjugate	(num -> num)	
cos	(num -> num)	Coseno
cosh	(num -> num)	
current-secconds	(-> entero)	
e	(-> real)	"e" es Euler: 2.71828182845
even?	(int -> bool)	
exact->inexact	(num -> num)	
exact?	(num -> bool)	Nos dice si un número es exacto o no
exp	(num -> num)	
expt	(num num -> num)	Eleva un número a determinada potencia
floor	(real -> int)	Aproxima a piso
gcd	(int int … -> int)	Máximo común divisor
imag-part	(num -> real)	
inexact->exact	(num -> num)	
inexact?	(num -> bool)	
integer->char	(int -> char)	
integer-sqrt	(num -> int)	
integer?	(any -> bool)	Nos dice si algo es un numero entero o no
lcm	(int int … -> int)	
log	(num -> num)	Calcula el logaritmo de un numero
magnitude	(num -> real)	
make-polar	(real real -> num)	
make-rectangular	(real real -> num)	
max	(real real… -> real)	Evalúa dos o más números y nos dice cuál es el mayor de todos
min	(real real… -> real)	Evalúa dos o más números y nos dice cuál es el menor de todos
modulo	(int int-> int)	Devuelve el módulo de la división entre dos números
negative?	(num-> bool)	Evalúa un número y dice si es negativo o no
number-	(num -> bool)	

>string		
number?	(any -> bool)	Evalúa un valor y nos dice si es un numero
numerator	(racional -> int)	
odd?	(int -> bool)	
Pi	-> real	"pi" es igual a: 3.141592653589
positive?	(num -> bool)	
quotient	(int int -> int)	Devuelve el cociente de la división entre dos números
random	(int -> int)	Devuelve un número aleatorio
rational?	(any -> bool)	
real-part	(num -> real)	
real?	(any -> bool)	Evalúa un valor y nos dice si es real
remainder	(int int -> int)	Devuelve el residuo de la división entre dos números
round	(real -> int)	Redondea un numero
sgn	(real -> union)	
sin	(num -> num)	Devuelve el seno de un numero
sinh	(num -> num)	
sqr	(num -> num)	
sqrt	(num -> num)	Calcula la raíz cuadrada de un numero
sub1	(num -> num)	
tan	(num -> num)	Devuelve la tangente de un numero
zero?	(num -> bool)	Evalúa un número y nos dice si es cero o no

3.2 Funciones Construidas (definidas por el usuario)

Al igual que en la mayoría de lenguajes de programación, en Racket podemos asociar valores a nombres, lo haremos usando la función *define*.

Muchas veces cuando programamos necesitamos usar un mismo valor en reiteradas ocasiones, con lo que no solo se hace ineficiente tener que escribirlo cada vez, tambien si tuviéramos que cambiar dicho valor tendríamos que hacerlo en todas las secciones de código donde aparezca. Estos inconvenientes pueden solucionarse asociando valores a nombres.:

Ejemplos:

- El número 3,1415 es equivalente a Pi: `(define Pi 3.1415)`
- La gravedad es equivalente a 9.8: `(define Gravedad 9.8)`

Ahora, cada vez que nos refiramos a Gravedad, Racket lo tomara como 9.8, de igual forma cada vez que nos refiramos a Pi, Racket lo remplazará con 3.1415.

Luego de haber hecho esto podremos realizar operaciones como:

```
(+ Pi Gravedad)      ; -> 12.9415
```

También resultaría muy útil en caso de que tuviéramos que cambiar el valor de Gravedad por un número más exacto, ya que no tendríamos que buscar todas las partes del codigo donde se encuentre el número 9.8 y remplazarlo por 9.799, solo tendríamos que modificar el valor que

anteriormente le asignamos a Gravedad:

```
(define Gravedad 9.799)
```

Recordemos que toda función tiene: nombre, entradas, argumentos y salida. Si analizamos los anteriores ejemplos nos daremos cuenta que también hemos estado creando funciones.

Para verlo mejor analicemos este ejemplo:

```
(define (Sumar arg1)
      (+ arg1 3))
```

Esta función posee: nombre: "Sumar", una entrada (parámetros): "arg1" y cuerpo:(+ arg1 3).

La asignación de valores que realizamos anteriormente (Pi, Gravedad) no fue más que la creación de funciones muy sencillas, es deducible que si toda función arroja un valor esta se puede sumar con otras funciones.

```
(+ (Sumar Gravedad) Pi)  ; -> 15.9405
```

Nota: El nombre de la función y de los argumentos los decide el programador, sin embargo es necesario ser elocuente al nombrar una función, por ejemplo si nuestro objetivo es obtener un promedio, deberíamos llamar a nuestra función: "promedio".

En la medida que avancemos por esta guía tendremos que construir nuestras propias funciones que serán cada vez más complejas.

3.3 LLAMADO A FUNCIONES

Para llamar una función basta con escribir el nombre de esta y remplazar los argumentos con valores operables.

Llamemos la función que declaramos anteriormente `(Sumar arg1)`, solo basta escribir su nombre y remplazar los argumentos (arg1) con valores operables, tal como lo haríamos con una función matemática que sería equivalente a *f(x)=x+3,* si introdujéramos 2 a la función tendríamos: *f(2)=5,* de la misma forma si introdujéramos 2 en nuestra función tendríamos:

```
(Sumar 2) -> 5
```

Forma general:

`(nombre-de-la-funcion arg1 arg2 arg3 …)` donde arg1 arg2 arg3… son los parámetros con los que operara la función.

Llamemos las funciones que creamos en los tres ejemplos anteriores:

```
Gravedad    ;->9.799
(Sumar 90) ;-> 93
(* 20 2)    ;-> 40
Pi          ;-> 3,1415
```

Nota importante: Racket diferencia entre mayúsculas y minúsculas, es por eso que si definimos una función como "Suma" esta solo podrá llamarse como "Suma" y nunca como "suma" o

“sUmA”.

3.4 Actividades

1. Experimenta con algunas de las funciones primitivas enlistadas en el cuadro de las páginas 21-22-23-24, trata de descubrir por ti mismo que hacen y cómo funcionan.
2. Crea funciones que asocien valores de constantes matemáticas o físicas a nombres, como por ejemplo: Euler, Pi, Gravedad, Velocidad, Kilogramo… (Al nombrar estas funciones usa palabras que describan el resultado de la función, empezar con letras en mayúsculas no es una regla general pero es bueno hacerlo por convención).
3. Llama a las funciones que creaste anteriormente y trata de operarlas con funciones primitivas, como: `"+", "*", "number?", "floor"…`
4. Experimenta con los tipos de datos que vimos al final del capítulo anterior, suma números enteros, reales y racionales, declara caracteres y símbolos, trata de identificar por ti mismo los errores que cometas.
5. Añade comentarios descriptivos a todos los ejercicios que realizaste.

4

EXPRESIONES ARITMÉTICAS, LÓGICAS Y DE COMPARACIÓN

En este capítulo estudiaremos algunas de las funciones primitivas que nos permiten realizar operaciones aritméticas y de tipo lógico, analizaremos algunos problemas y les daremos solución utilizando dichas funciones y otras creadas por nosotros mismos.

4.1 OPERADORES ARITMÉTICOS

¿Qué es una expresión aritmética?: Se entiende por expresión aritmética a aquella donde los operadores que intervienen en dicha expresión son numéricos, el resultado es un número y los operadores son aritméticos.

Los operadores aritméticos ya los conocemos:

+	-	*	/
Suma	Resta	Multiplicación	División

En Racket, estos operadores aparecen como funciones primitivas y se representan con su símbolo correspondiente, tenemos muchas funciones para operar números como vimos en el capítulo anterior 3.1.1:

Para realizar cualquier operación aritmética esta debe ir entre paréntesis y se coloca el operador en primer lugar, los números se ubican después del operador y se separan por espacios (Recordar la notación Preorden, capitulo 1.4).

Ejemplo:

```
(*(+ 2 2) (/(*(+ 3 5) (/ 30 10))2))
```

4.2 PROBLEMAS CON SOLUCIÓN MATEMÁTICA

Ejercicio N° 1:

Crearemos una función que calcule la distancia entre dos puntos de una recta.

La fórmula general para esta operación es: $\sqrt{(x2 - x1)^2 + (y2 - y1)^2}$

El código de esta función quedaría así:

```
1 ;Definimos "Distancia_puntos" y sus argumentos
2 (define (Distancia_puntos X1 X2 Y1 Y2)
3   (display "Distancia: ") ;Imprime por pantalla "Distancia: "
4   ;Ecuación de la distancia expresada en Preorden
5   [sqrt (+(sqr(- X2 X1))(sqr(- Y2 Y1)))]
6 );Cerramos la función
```

En el ejercicio anterior se observa la vital importancia de los paréntesis en Racket ya que estos le indican al intérprete en qué orden realizar cada operación y en donde empieza y donde termina.

Usamos funciones muy sencillas como “sqrt” (raíz cuadrada), “sqr” (elevar al cuadrado), sumas y restas. La función “display” muestra por pantalla el texto que esta entre comillas.

Como vimos en el capítulo anterior para operar con esta función es necesario llamarla (escribir su nombre y remplazar con valores operables los argumentos).

Ejercicio N° 2:

Calcular el área de un anillo: El área de un anillo se calcula restando el disco exterior del disco interior, lo que significa que nuestro programa requiere de dos cantidades desconocidas, el radio exterior y el radio interior, a esos números los llamaremos como R-ext y R-int.

En nuestro pequeño programa, al igual que en el 99% de los programas que realicemos, debemos crear más de una función. Nuestros programas se deben dividir en varias funciones que se llamen unas a otras para hacer que el código sea más legible, para que el programa tenga un mayor rendimiento, además, dividir un problema en funciones ayuda a resolver este más fácilmente e incluso facilita la detección y corrección de errores.

```
1  (define (AreaAnillo R-int R-ext)
2    ;Area = (Pi*r²)-(Pi*r²)
3    (- (* pi(* R-extR-ext)) (* pi(* R-intR-int)))
4  )
5  ;Definimos la función para mostrar en pantalla el área
6  (define (Area var1 var2)
7    (display "Área del anillo: ")
8    (display (AreaAnillo var1 var2)) )
```

El llamado de la función quedaría así:

`(Area 2 6)` ; Donde 6 es el radio exterior y 2 el radio interior.

Ejercicio N° 3:

Se necesita calcular el promedio de las notas de un estudiante de colegio, el sistema de calificaciones está definido por los siguientes típicos: Durante el periodo de estudio se sacan 4 notas (a, b, c, d). Las calificaciones se pueden dar en números decimales y cada una vale un porcentaje específico, así: la nota “a” vale 50%, “b” vale 20%, “c” vale 20% y “d” vale 10%. La calificación menor es 1 y la mayor es 5. Calcular la nota final del estudiante teniendo en cuenta que esta no puede ser un numero decimal, debe ser un entero comprendido entre 1 y 5.

```
1  (define (Promedio A B C D)
2    (+ (* A 0.50) (* B 0.20) (* C 0.20) (* D 0.10)))
3  (define(PrintPromedio a b c d)
```

```
4    (display "El promedio total y redondeado es: ")
5    (round (Promedio a b c d))
6  )
7   (PrintPromedio 30 3 2.1 3)
```

4.3 Operadores Lógicos y de comparación

Los operadores lógicos y de comparación nos proporcionan un resultado a partir de que se cumpla o no una cierta condición.

¬	^	v	=	≠	>	<	≥	≤
No (not) Lógico	Y (and) Lógico	O (or) Lógico	Igual	Diferente	Mayor que	Menor que	Mayor o igual que	Menor o igual que

- ¬ No lógico: Se usa para negar una proposición, ejemplo: '¬A' significa todo lo que no es A.

  En Racket se representa con la función *"not"*.

- ^ Y lógico: Se usa para evaluar dos expresiones, ejemplo: 'A ^ B' devuelve verdadero si y solo si A y B cumplen una condición, de lo contrario devuelve falso.

  En Racket se representa con la función *"and"*.

- v O lógico: Se usa para evaluar dos expresiones, ejemplo 'A v B' devuelve verdadero si A, o B cumplen una condición, en caso de que, ni A, ni B la cumplan devolverá falso.

  En Racket se representa con la función *"or"*.

- = Igual: Se usa para comprar una expresión con otra, si estas son iguales devuelve verdadero, de lo contrario falso.

  En Racket se representa con el símbolo *"="* pero este únicamente compara valores numéricos, si lo que queremos comprar son caracteres, símbolos, cadenas etc… debemos usar la función "equal?".

- ≠ Diferente: Se usa para comparar dos expresiones, si estas son diferentes devuelve verdadero, de lo contrario falso.

  En Racket no existe una función definida que signifique diferente, pero podemos lograrla usando dos operadores lógicos: no e igual, de esta forma negando una igualdad conseguimos el mismo resultado.

  Ejemplo: `(not (= 3 4))` *;-> Verdadero*

- > Mayor que: Se usa para comprar dos números, si el primero es mayor que el segundo devuelve verdadero, de lo contrario falso.

  En Racket se representa con el símbolo: >

- < Menor que: Se usa para comparar dos números, si el primero es menor que el segundo devuelve verdadero, de lo contrario falso.

  En Racket se representa con el símbolo: <

- ≥ Mayor o igual que: Se usa para evaluar dos números, si el primero es mayor o es igual al segundo devuelve verdadero, de lo contrario falso.

  En Racket se representa con el símbolo: >=

- ≤ Menor o igual que: Se usa para evaluar dos números, si el primero es menor o es igual al segundo devuelve verdadero, de lo contrario falso.

  En Racket se representa con el símbolo: <=

Nota: DrRacket devuelve "#t" como verdadero y "#f" como falso.

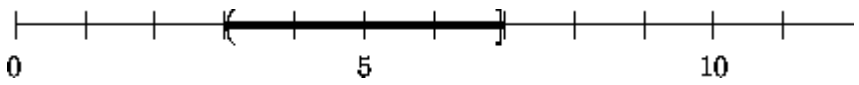
4.4 ACTIVIDADES

1. Realice las siguientes operaciones con números en DrRacket.
2. Calcule el valor absoluto de un número
3. Elevar un número a una potencia dada.
4. Calcule la raíz cuadrada de un número
5. Calcule el residuo de una división entera
6. Calcule el logaritmo de un número
7. Calcule el número más grande entre 5 números
8. Calcule el número más pequeño entre 5 números
9. Genere un número aleatorio
10. Redondee un número
11. Cree una función que calcule el volumen de un cilindro: $\pi * r^2 * h$
12. Cree una función que calcule el volumen de una esfera : $4/3 * \pi * r^3$
13. Cree una función que calcule el volumen e un cono: $1/3 * \pi * r^2 * h$
14. Los contadores siempre usan programas que calculan los impuestos basados en el pago de las personas. El impuesto que tienen que pagar es del 15% del pago que hagan las personas. Hacer un programa que calcule dicho impuesto.
15. El supermercado "El Ahorrito" necesita un programa que calcule el valor de una bolsa de monedas. Defina un programa que reciba el número de monedas de: $20, $50, $100, $200 y $500 que hay en la bolsa y devolver la cantidad de dinero que hay en ella. (Asumir que el primer parámetro corresponde a la cantidad de monedas de $20, el segundo a las de $50, y así sucesivamente)
16. Se desea hacer un programa que calcule las ganancias de un teatro para una presentación. Cada cliente paga $10.000 por entrada y cada función le cuesta al teatro $300.000 por la atención prestada y por cada cliente que entre, el teatro debe pagar un costo de $2.000 por aseo. Desarrolle un programa que reciba el número de clientes de una función y devuelva

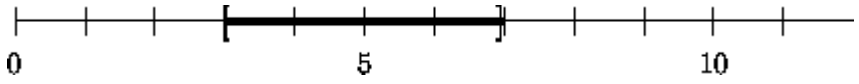
el valor de las ganancias obtenidas.

17. Realice las siguientes operaciones con expresiones lógicas en DrRacket.
    a. Hacer una función que reciba un número y devuelva verdadero o falso si es igual a 5.
    b. Hacer una función que reciba un número y devuelva verdadero o falso si el número es mayor o igual que 10.
    c. Hacer una función que reciba un número y devuelva verdadero o falso si el número es menor que 20.
    d. Hacer una función que reciba un número y devuelva verdadero o falso si el número es mayor o igual que 10 y menor que 20.
    e. Hacer una función que reciba un número y devuelva verdadero o falso si el número es mayor o igual que 10 y menor que 20 o es mayor que 30

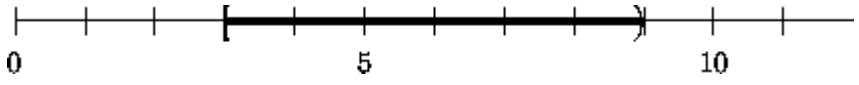
18. Traslade los siguientes 4 intervalos de la línea recta a funciones de Racket que acepten un número y retornen "True" si el número está en el intervalo y "False" si está afuera de él:
    a. El intervalo (3,7]:



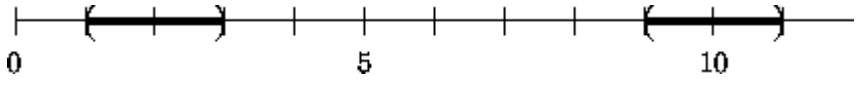
    b. El intervalo [3,7]:



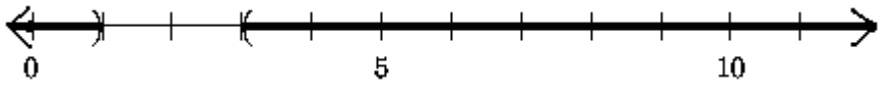
    c. El intervalo [3,9):



    d. La unión de (1,3) y (9,11):



19. El rango de números que no pertenezcan a [1,3]:



5

CONDICIONALES

¿Expresiones condicionales?

Una expresión condicional es una instrucción o grupo de instrucciones que se pueden ejecutar o no, depende de si dicha condición se cumple o no se cumple.

A diferencia de los operadores lógicos o de comparación que vimos en el capítulo anterior, en los que dependiendo de la veracidad de una sentencia se devolvía un verdadero *(#t)* o falso *(#f)*, con los condicionales, dependiendo de la veracidad de la sentencia que evaluemos se ejecutara una instrucción, o se ejecutara otra de ser falsa. (En el paradigma funcional, y más precisamente en Racket, las sentencias de comparación y las expresiones condicionales están estrechamente ligadas.)

Veamos cuales son las Expresiones Condicionales:

if	cond

5.1 BOOLEANOS Y SENTENCIAS DE COMPARACIÓN

Los Booleanos y las sentencias de comparación están estrechamente ligados a los condicionales, cuando queremos crear una condición esta evalúa sentencias de comparación que arrojan como resultado un booleano y dependiendo de este se ejecutara una instrucción u otra.

Recodemos de lo visto en el capítulo anterior que Racket tiene palabras reservadas para expresar verdad y falsedad, estas son *#t* y *#f* respectivamente. Si declaramos una sentencia de comparación (o de relación) el resultado de esta se expresara en #t o #f que son lo que conocemos como booleanos.

- `(= x y) ;"¿x es igual a y?"`
- `(< x y) ;"¿x es menor que y?"`
- `(> x y) ;"¿x es mayor que y?"`

Veámoslo de la siguiente manera: cuando introducimos una sentencia de comparación en Racket, le estamos preguntando, por ejemplo si x es igual a y, a lo que él nos responderá sí o no (*#t*, *#f*) luego, si tenemos un condicional, se ejecutara una sección de código en caso de cumplirse lo que le preguntamos. Si bien no siempre es necesario que exista una sección de código a ejecutar en caso de que no se cumpla lo que preguntamos, es recomendable que si exista.

5.2 EL CONDICIONAL: IF

Esta es su forma general:

```
(if (condicion)
    (Respuesta_Verdadero)
    (Respuesta_Falso)
)
```

Como dijimos anteriormente, la respuesta para el caso falso no siempre es necesaria. (Solo si programamos en lenguaje "Muy grande", pero, si cambiamos a, por ejemplo, "Estudiante avanzado" es obligatorio incluir la parte falsa, de lo contrario arrojara un error y el código no se ejecutara)

Creemos una pequeña función que nos diga si el resultado de una suma es mayor a 10:

```
(define (Suma N1 N2)
  (+ N1 N2)
)
(define (MayorQ10? N1 N2)
  (if (> (Suma N1 N2) 10)
      (display "Si es mayor que 10")
       (display "No es mayor que 10")
   )
)
(MayorQ10? 2 3) ;-> "No es mayor que 10"
```

Fue una función muy sencilla, pero podría pulirse más en muchos aspectos, por ejemplo, en caso de que N1 y N2 sumen más de 10 se imprimiría por pantalla "Si es mayor que 10"… pero, que es mayor que 10? Sería más correcto imprimir el numero también, así: "N1 + N2 Es mayor que 10".

Otro aspecto es el siguiente: No es necesario que creemos una función para sumar podemos hacerlo directamente. Veamos el código del mismo ejemplo pero con estos dos aspectos mejorados.

```
(define (MayorQ10? N1 N2)
  (if (> (+ N1 N2) 10)
     {begin
        (display (+ N1 N2))
        (display ", Es mayor que 10")
       }
     {begin
        (display (+ N1 N2))
       (display ", No es mayor que 10")
       }
   ) )
 (MayorQ10? 2 3) ;-> "5, No es mayor que 10"
```

Luego de analizar el código se habrá dado cuenta del uso de una nueva función: "begin"; Si hubiéramos escrito el código sin esta función, Racket no encontraría cuál de todas las líneas de código es la correspondiente para la respuesta verdadera o para la falsa. Es por eso que usamos la instruccion "begin", para *agrupar* el código en partes, así Racket vera estos dos grupos de código y como es normal ejecuta el primer grupo (primer begin) de cumplirse la condición o la segunda de lo contrario.

Qué tal si le introducimos al programa 0 y 10, su suma es 10, por lo que nos imprime: “10, No es mayor que 10”, pero sería mucho más acertado imprimir “10, es igual a 10”. Hagámoslo:

```
(define (MayorQ10? N1 N2)
  (if (> (+ N1 N2) 10)
    {begin
      (display (+ N1 N2))
      (display ", Es mayor que 10")
     }
    (if (= (+ N1 N2) 10)
      {begin
        (display (+ N1 N2))
        (display ", Es igual a 10") }
          {begin
            (display (+ N1 N2))
            (display ", No es mayor que 10")
       }
     )
   )
)
(MayorQ10? 2 3) ;-> "5, No es mayor que 10"
```

A esto se le llama ***anidamiento,*** un if dentro de otro if, y podríamos incluir cuantos más necesitáramos, uno dentro de otro. En nuestro sencillo programa el segundo if es la parte falsa del primero, si la suma de dos números no es igual a 10 entrara en esta segunda sentencia condicional que evalúa si la suma de los dos números es igual a 10, de serlo imprime que son iguales, de lo contrario se deduce que si no es igual y ni mayor que 10 entonces es menor.

5.3 El condicional: cond

Su forma general:

```
(cond
  [(condición1) (Respuesta_verdadero)]
  [(condicion2) (Respuesta_verdadero)]
  . . .
  [(condicionN) (Respuesta_verdadero)]
  (else (Respuesta_falso) )
)
```

Al usar “cond”, independientemente de si nuestro lenguaje es “Muy grande” o “Estudiante

avanzado" la parte falsa es opcional, se escribe luego todas las condiciones y únicamente se ejecuta si ninguna de las condiciones anteriores es verdadera.

No dejemos de lado nuestro ejemplo anterior, hagámoslo ahora usando "cond".

```
(define(MayorQ10? N1 N2)
  (cond
    [(>(+ N1 N2)10) (begin (display(+ N1 N2)) (display " Es Mayor que 10"))]
    [(=(+ N1 N2)10) (begin (display(+ N1 N2)) (display " Es Igual que 10"))]
    [(<(+ N1 N2)10) (begin (display(+ N1 N2)) (display " Es Menor que 10")) )
  ) )
```

Nuestro código podría llevar "else" (que se ejecuta en caso de que ningúna de las condiciones anteriores se cumpla) sin alterar el resultado:

```
(define(MayorQ10? N1 N2)
  (cond
    [(>(+ N1 N2)10) (begin (display(+ N1 N2)) (display " Es Mayor que 10"))]
    [(=(+ N1 N2)10) (begin (display(+ N1 N2)) (display " Es Igual que 10"))]
(else (begin (display (+ N1 N2)) (display " Es Menor que 10")) )
  ) )
```

Es notablemente mucho más ventajoso usar "cond" en vez de "if" en este caso, tal cual habrán casos en los que el uso de if será más adecuado que cond, no solo depende de nuestro criterio, pues habrán ocasiones en las que obligatoriamente tendremos que usar if, ya que con el podemos lograr resultados que difícilmente se podrían alcanzar usando cond.

Es importante tener en cuenta al usar cond: Cuando se encuentra una condición que resulta cumplirse, se deja de evaluar todo lo demas y finaliza la ejecución de dicho cond.

ELSE, se usa para declarar la parte falsa de todo el condicional, siempre debe declararse al final del condicional. Luego de Else no puede ir una sentencia de comparación, solamente puede ser un resultado, un display u otro condicional.

5.4 Ejemplos de programas con Condicionales

Ejercicio Nº 1:

Suponga que el banco paga 4% para depósitos menores a $50.000 (inclusive), 4.5% para depósitos menores a $250.000 (inclusive), y 5% para depósitos mayores a $250.000. Realice una función que reciba un depósito y devuelva el porcentaje asignado.

La respuesta a este problema es fácil, y la encontraremos aún más fácil si usamos el condicional "cond", si usáramos "if" podríamos llevarnos más líneas de código que las necesarias.

```
1 (define (interés cantidad)
2   (cond
3     [(<= cantidad 50000) 0.040]
```

```
    [(<= cantidad 250000) 0.045]
    [(> cantidad 250000) 0.050]))
```

Si aplicáramos a “interés” una cantidad, por ejemplo, 70000, Racket procedería a reemplazar cantidad con el valor que le indicamos y luego realizaría las operaciones que escribimos en el cuerpo de la función, finalmente tendríamos que la primera condición es False, pero la segunda es True, por lo que el resultado sería: 0.045.

Ejercicio Nº 2:

Crear una función en la que el usuario ingrese un número y el programa devuelva si dicho número se encuentra entre 1 y 3, de lo contrario imprima que el número no está en el intervalo 1-3.

```
(define (leer)
  (begin
    (display "Ingrese un numero: ")
    (read)
  ) )
(define (Intervalo a)
  (cond
    [(and (<= a 3)(>= a 1))
     "El número se encuentra en el intervalo 1-3"]
    (else "El número no se encuentra en el intervalo 1-3")
  ) )
 (Intervalo (leer))
```

Esta es una función realmente sencilla, pero que ejemplifica a la perfección el uso del operador lógico “and”. Analicemos un poco: Bien se hubiera podido usar el condicional if para verificar que el número introducido es mayor que uno, luego en la parte verdadera introducir otro if (anidamiento) para que verifique que el número es menor que tres, de ser correctas las dos condiciones imprimir en pantalla que el número se encuentra en nuestro intervalo o de lo contrario imprimir que no lo está, pero para hacer menos complejo y extenso nuestro código optamos por usar “cond” y el operador lógico “and” que hace que se evalúen las dos condiciones y solamente si ambas son verdaderas imprime que el número se encuentra en el intervalo y si alguna de las dos no se cumple imprimir que no lo está.

Nuestro pequeño programa tiene dos funciones, la segunda es, evidentemente, quien se encarga de verificar si el número que introdujimos se encuentra o no en el intervalo, pero la primera función aun no nos es familiar. Esta función imprime “ingrese un numero: ”y a continuación lee un numero por teclado y lo imprime en pantalla, pero como es que la función “Intervalo” puede tomar a “leer” como argumento e interpretar la entrada como valor para evaluar ignorando el texto en pantalla “ingrese un numero”?, esto es porque internamente el resultado de una función nunca será un “display”, será únicamente los valores que arrojen otras funciones diferentes a display, o que nosotros mismos indiquemos, por lo que “read” es el valor que toma la función “Intervalo” para operar.

Ejercicio Nº 3:

Queremos jugar Doble o nada, así que vamos a crear un programa que nos permita hacerlo, pero primero vamos a plantear cómo será el juego:

Para jugar doble o nada necesitamos dinero y una moneda, supongamos que queremos apostar un dólar, entonces tiramos la moneda, si cae cara, tu ganas todo, si cae sello lo pierdes todo.

En ese orden de ideas vamos a crear el programa. Lo primero será hacer una función que nos pregunte cuanto queremos apostar, luego otra función que "tire la moneda" y por último la función principal que nos diga si ganamos o perdimos.

Pero antes de empezar a programar analicemos un poco más a fondo el problema, pues esa no es la única forma en la que podemos hacer nuestro juego, bien podríamos usar una sola función. También debemos tener en cuenta que el juego debe ser imparcial o legal, en otras palabras no podemos ser nosotros quienes decidamos que lado de la moneda cae ya que no tendría sentido. Veamos el porqué de usar tres funciones y como *"Tirar la moneda imparcialmente",* veamos el código:

```
(define (Jugar Apta DbNd)
  (if (equal? DbNd "Doble")
      (begin
        (display "Ganas! tienes el doble: ")
        (display (+ Apta Apta)) )
      (display "Perdiste todo!!!!!!!!")
      )
  )
(Jugar (Apuesta) (DobleNada))
```

La función "random x" nos arroja un numero aleatorio cuyo valor se encuentra entre 0 y x-1 lo que quiere decir que si realizamos "random 2" la función nos devuelve uno o cero con lo que se soluciona el problema de la moneda, ya que esta solo tiene dos lados podemos llamar a un lado 0 en vez de cara y al otro 1 en vez de sello.

¿Porque usar 3 funciones y no una? La mayor utilidad de una función es su reutilización, si en un programa tenemos que usar una formula o realizar determinado procedimiento muchas veces sería muy poco práctico e incómodo tener que escribir cada vez dicha fórmula o procedimiento, en cambio podemos crear una función y reutilizarla cuantas veces lo necesitemos, además la mejor forma de resolver un problema es dividirlo, en nuestro caso dividirlo en funciones, es por eso que usamos tres funciones en vez de una, por si luego queremos hacer nuestro juego más complejo podamos reutilizar nuestras funciones y para dar un orden más lógico a nuestro programa de manera que sea más fácil de entender

5.5 Actividades

1) Hacer una función que reciba un número. Si el número es mayor o igual que diez imprima por pantalla "El número es mayor o igual que 10", si es menor que 10 muestre por pantalla "El número es menor que 10".
2) Un almacén de venta de harina opera bajo el siguiente esquema de venta: si le compran hasta

10kg de harina vende la cantidad exacta que le soliciten, si le compran hasta 100 kg de harina encima 3kg mas, si le compran hasta 500 kg de harina encima 10kg mas y si le compran más de 500 kg de harina encima el peso equivalente al 10% del valor solicitado. Construir una función que permita calcular la cantidad de harina a entregar a partir de la cantidad solicitada. (Nota: El programa debe pedir que el usuario ingrese la cantidad de harina a comprar, luego el programa devuelve la cantidad de harina que entregara incluyendo la encima si es que la tiene.)

3) Se va realizar un paseo en una escuela. Si van hasta 10 niños, el valor por cada niño es de 3000 pesos. Si van más de 10 niños pero menos o igual que 50, el valor por cada niño es de 2500 pesos. Si van más de 50 niños pero menos o igual que 200, el valor por cada niño es de 2000. Si van más de 200 niños, el valor es de 1800 pesos. Construir una función que permita calcular el dinero en total que se va a recolectar para el paseo dependiendo de la cantidad de niños que vayan a ir.

4) Un vendedor puerta a puerta debe trabajar bajo las siguientes normas: si en el día recorre menos de 10 calles debe realizar por lo menos 20 ventas, si en el día recorre entre 11 y 30 calles entonces debe realizar por lo menos 60 ventas, si en el día recorre entre 31 y 100 calles entonces debe realizar por lo menos 80 ventas. Por cada venta que haga se le pagará un valor de 3000 pesos y, cuando se totalice la venta, se le debe descontar el valor equivalente al 16% de la venta total por razones legales. Construir una función que permita determinar el valor a pagarle a un vendedor que trabaja con esta tabla de ventas. (Nota: El programa debe solicitar el número de calles recorridas)

5) Todas las ecuaciones cuadráticas (en una variable) tienen la siguiente forma general: $a*x^2+b*x+c= 0$

   El número de soluciones de una ecuación cuadrática depende de los valores de a, b y c. si el coeficiente de a es 0, podemos decir que la ecuación es degenerada y no se consideran cuantas soluciones tiene. Asumiendo que a no es 0, la ecuación tiene

   i) dos soluciones si $b^2 > 4 \cdot a \cdot c$,

   ii) una solución si $b^2 = 4 \cdot a \cdot c$, and

   iii) ninguna solución si $b^2 < 4 \cdot a \cdot c$.

   Desarrollar la función Cuantos, la cual recibe los coeficientes a, b y c de una ecuación cuadrática apropiada y determine cuantas soluciones tiene la ecuación:

   (Cuantos 10-1) =2

   (Cuantos 242) =1

6. Escriba un programa que reciba un numero e indique si se trata de un numero par

7. Hacer un programa que dado el día (número), el mes (número) y el año (número), el retorne el día (lunes, martes…) de la semana a la que pertenece la fecha ingresada

8. Desarrolle la función Impuesto, la cual recibe como parámetro el pago total y devuelve la cantidad de impuestos ganados. Para un pago de $400,000 o menos, el impuesto es del 0%; para un pago entre $400,000 y $800.000, el impuesto es del 15%; y para un pago mayor o igual a $800,000, el impuesto es del 28%. También desarrolle la función

PagoNeto. La función determina el pago neto de un empleado basado en el número de horas trabajadas. Asuma que se paga a $20.000 por hora. El pago neto es el pago total menos el impuesto. Utilizar la función Impuesto para calcular el pago neto.

Explicación: Pago total = número de horas trabajadas * 20000

Pago neto = Pago total - Impuesto

Nota: Recuerde desarrollar funciones auxiliares cuando una definición es muy grande o compleja de manejar. Use las funciones “display”, “newline” y presente texto por pantalla de forma que el programa sea intuitivo y amigable con el usuario.

9. Cree un programa que devuelva varias propiedades de un numero dado, si el usuario ingresa x número El programa deberá devolver si el número es par o impar, natural o entero, exacto o inexacto, cero o diferente de cero, además, si el usuario ingresa un carácter en vez de un numero el programa debe imprimir por pantalla: “Usted ingreso letras, pero este programa solo funciona con números”.

   Nota: Para que el programa funcione correctamente es indispensable decidir bien entre usar “cond” o “if”. Analice porque.

6

RECURSIVIDAD

6.1 QUE ES UN BUCLE

En programación, un bucle es una sentencia que se realiza repetidas veces hasta que la condición asignada a dicho bucle deje de cumplirse.

Ejemplo: Mientras *x* sea menor que 10 Sumar 1 a *x*

Según el ejemplo anterior, si tenemos que: x = 0, 0 es menor que 10, por lo que le sumamos 1 a 0 y tenemos que x = 1, nuevamente 1 es menor que 10, por lo que le sumamos 1 y tenemos que x=2. Así sucesivamente hasta que x sea 10, solo entonces no se sumaran más valores a x, dado que 10 no es menor que 10, y el bucle finalizara.

La mayoría de lenguajes de programación tienen funciones para realizar bucles, por ejemplo en lenguaje C se usa: "while", pero en Racket por ser un lenguaje funcional, no contamos con bucles como tal, -estos tendrán la oportunidad de ser estudiados en otro curso de programación más avanzado- a cambio tenemos un método de igual utilidad: La recursividad.

6.2 RECURSIVIDAD/ITERACIÓN

¿Qué es la recursividad?: Es la forma en la cual se especifica un proceso basado en su propia definición.

¿Qué es la Iteración?: En programación, es la repetición de una serie de instrucciones en un programa de computadora.

La diferencia de estos dos términos depende más del contexto en que los usemos, así que para no entrar en discusiones y con el fin de facilitar el aprendizaje diremos que la una implica la otra, y en el resto del libro hablaremos siempre de recursividad.

Finalmente podemos decir que: En programación la recursividad es un concepto básico que corresponde a un proceso en el cual una función se llama a si misma (y por tanto esta función se ejecuta N número de veces) pudiendo provocar un llamado infinito. Como programadores al diseñar una función recursiva debemos evitar que esta se llame infinito número de veces y darle *un caso base* en el cual la función dejara de llamarse a sí misma.

Para entender la recursividad más fácilmente imaginemos que estamos parados frente a un espejo (espejo 1) mientras sostenemos otro en nuestras manos (espejo 2), Si miramos en el espejo que tenemos en frente (el 1) el reflejo del espejo que sostenemos en nuestras manos veremos la misma imagen repetirse una y otra vez. Este ejemplo representa un caso de recursión infinita, algo que nunca debemos hacer cuando programemos.

Veamos un último ejemplo muy usado para explicar la recursividad en el que también se enumeran las normas básicas que debemos aplicar al usar la recursividad en nuestros programas:

1. Si ya entendiste el concepto de recursividad, salta al siguiente parrafo, si aún no lo entiendes continua leyendo.
2. En cada paso recursivo (o ciclo, o loop) debes avanzar un poco en algo que te diga explícitamente cuantas recursiones has hecho y que progreso has logrado (cuan cerca

estas de la solución al problema por el que has creado la función recursiva).

3. Además de avanzar, debes hacer una acción, con cada paso debes estar más cerca de solucionar el problema.
4. Toda función recursiva debe llamarse a sí misma. Esa es la clave de la recursión. Como esto no es un lenguaje de programación lo simularemos simplemente diciéndote que vuelvas a leer desde el paso 1.

Recuerda que siempre debes tener un *caso base* que te diga cuándo parar, en el ejemplo anterior nuestro caso base era entender el concepto.

Cabe anotar que: Toda función que se llama a si misma se denomina Recursiva, tendremos que diseñar almenos un caso base y casos generales:

Caso base: Son los casos del problema que se resuelven sin recursividad. Siempre debe existir almenos un caso base y este debe proporcionar una solución o ayudarnos a llegar a ella.

Casos generales: Son los casos que siempre deben avanzar hacia un caso base.

Hay dos elementos esenciales a tener en cuenta a la hora de crear una función recursiva en Racket: el condicional y el llamado de la función a sí misma. Si recordamos el ejemplo de bucle al inicio de este capítulo vemos que existe una condición, esta es quien dice cuando termina (o cuando continua) de ejecutarse el código, o en el caso preciso del ejemplo cuando hayamos entendido el concepto pararemos de leer.

¿Cómo logramos eso computacionalmente?: sencillo, el lenguaje de programación Racket nos proporciona una herramienta muy útil llamada: *expresión condicional,* este siempre tendrá código a ejecutar cuando la condición se cumpla y otro segmento de código cuando esta no se cumpla, dependiendo de cómo y de qué queremos hacer podemos incluir el caso base en la parte falsa o en la parte verdadera pero sea cual sea el lugar que elijamos en el contrario deben encontrarse los casos generales y el llamado recursivo de la función (llamado recursivo es el segmento de código en el que la función se llama a si misma).

Sin más que explicar, vamos a la práctica:

6.3 Ejemplos básicos de recursividad

Ejercicio No 1:

Imprimir los números del uno al diez recursivamente:

```
(define (PrintNum-10 x)
  (if (= x 10) ;El caso base se dará cuando x sea igual a 10
    (display x) ;Cuando el caso base se cumpla se imprimirá "x"
      (begin ;Casos generales, cuando x no es igual a 10
        (display x) ;si x diferente de 10 imprimir "x"
        (PrintNum-10 (+ 1 x))) ;si x diferente de 10, volver a llamar la función estableciendo a "x" como x+1
  )
)
(PrintNum-10 0) ;Llamado de la función
```

La característica principal de la recursividad es que una función SE LLAMA A SI MISMA para incrementar o restar el valor de un parámetro (en este caso se incrementa el valor de x para que llegue a ser igual a 10 y así lograr que el procedimiento tenga un final, es decir que no se quede en un ciclo infinito).

Hagamos un paso a paso del programa:

- La línea 1 es la definición de la función, le hemos asignado por nombre "PrintNum-10" y tiene un parámetro o argumento que es "x".
- En la segunda línea entramos en materia: Como explicamos anteriormente en Racket no contamos con bucles, a cambio tenemos la Recursividad, por lo que siempre que queramos hacer uso de esta debemos tener en cuenta esos dos tópicos que convertirán nuestra función en una función recursiva: El condicional y el llamado de la función a sí misma. Usamos condicional: if, y nuestra condición es que "x" sea igual a 10 (= x 10), esta condición también nos define el caso base y los casos generales. (Caso base: cuando x sea igual a 10. Casos generales: Cuando x sea diferente de 10)
- La línea 3 le indica a Racket que hacer cuando se cumpla el caso base, (display x) imprimir x.
- La línea 4 es una función que ya habíamos visto, pero recordemos que hace: sencillamente begin nos permite agrupar más de una línea de código. En nuestro caso estamos agrupando dos líneas para que constituyan la parte falsa de nuestro condicional (o los casos generales de nuestra función recursiva).
- La línea 5, (display x) imprimir x. Esto se ejecuta únicamente si x es diferente de 10.
- La línea 6, el Llamado recursivo. `(PrintNum-10 (+ 1 x))` cuando Racket ejecute esta línea la función se ejecutara de nuevo, por lo que tenemos que regresar a leer la línea 1 pero esta vez el parámetro x será x+1.

Otro elemento a rescatar en esta línea de código es ver como en el llamado a una función podemos incluir otras funciones.

- Líneas 7 y 8: La primera cierra el condicional y la segunda cierra toda la función.
- La línea 9: es donde llamamos la función. Para el ejemplo hemos llamado la función inicializando x en 0, si ejecutamos el código tenemos en pantalla los números del 0 al 10, pero que pasaría si en vez de 0 llamamos nuestra función con 10?: simplemente se imprimiría en pantalla el número 10.

Ahora para practicar te recomiendo que hagas otra función para imprimir los números del 10 al 0.

Hacer esta prueba: cambiar el llamado (+ n 1) por (- n 1). Ejecutar y luego hacer debug. Recuerden que para parar un programa que se quedó bloqueado se usa la opción 

Ejercicio No 2:

Hacer un programa que calcule la cantidad de divisores de un número utilizando la recursividad:

Análisis: ¿Cómo calculamos la cantidad de divisores que tiene un número? Podemos dividir el número por sí mismo y por todos los números menores a él, cuando el resultado de dicha división sea un entero habremos encontrado un divisor. Este proceso resultaría casi imposible para un

humano, solo imaginemos encontrar los divisores de 11024000, tendríamos que dividir ese mismo número de veces, algo que nos tomaría una cantidad exagerada de tiempo. Pero para una computadora con un procesador actual es una tarea sencilla que no tomaría más de unos pocos milisegundos.

El código:

```
1  (define(Divs A Cont)
2    (if (= Cont 1) ;Caso base cuando cont = 1
3      (display (/ A Cont)) ;Imprimir el # divisor
4      (begin ;Casos generales, cuando cont no es igual a 1
5        (if (integer? (/ A Cont))
6          (begin
7            (display(/ A Cont))
8            (newline) ;Salto de linea
9            (Divs A (- Cont 1)); Llamado recursivo
10          )
11         (Divs A (- Cont 1));Llamado recursivo
12 ) ) ) )
13 (define (Divisores X)
14   (Divs X X) )
15 (Divisores 110240)
```

Análisis del código:

Tenemos dos funciones: (Divs A Cont) y (Divisores X).

(Divs A Cont) es la función que calcula los divisores de A, Cont es un "contador" que nos sirve de dividendo y que a la vez es quien nos dice cuántos llamados recursivos se deben hacer. Cont siempre será igual a A dado que si queremos hallar los divisores de 400 debemos dividirlo 400 veces o, en otras palabras: realizar 400 llamados recursivos.

En la línea dos está el condicional que nos indica el caso base: cuando Cont es igual a 1. Dado que en cada llamado recursivo Cont se disminuye en uno este llegara a ser 1 y habremos terminado de dividir.

La línea 3 corresponde a qué hacer en caso de cumplirse el caso base. Recordemos que el caso base se da cuando Cont es igual a uno y todo número divido entre uno da el mismo número por lo tanto habremos encontrado el ultimo divisor y también hay que imprimirlo

Para los casos generales (líneas desde 4 hasta 10): Si el resultado de dividir "A" entre "Cont" es un entero sin punto flotante (integer? (/ A Cont)) entonces habremos encontrado un divisor y hay que imprimirlo en pantalla (lina 7), luego hacer el llamado recursivo, pero esta vez estableciendo Cont como Cont menos uno (línea 9). En el caso que la división entre A y Cont no sea igual a un entero se hará otro llamado recursivo reduciendo a Cont en uno.

Las líneas 13 y 14 no son necesarias para que el programa funcione, pero sí lo son para presentar un programa amigable al usuario, puesto que si dejamos únicamente la primera función, al calcular los divisores de un número (200 por ejemplo) el llamado seria: (Divs 200 200), habría

que escribir dos veces el mismo número y esto le resta claridad al programa, mucho más en el caso de que lo use alguien diferente de quien lo programo. Es por eso que creamos la función (Divisores X). En donde llamamos a (Divs A Cont) y reemplazamos los dos parámetros A y Cont, con X.

Por ultimo en la línea 15 el llamado a la función. Si ejecutamos con el número del ejemplo: 11024000 podremos ver que realmente tarda pocos segundos en calcular los divisores.

Hay varios elementos a destacar en este pequeño programa: como lo son el uso de ifs anidados y más de un caso general en la recursión. (Cuando Cont diferente de 1 y cuando el resultado de a / Cont es o no un entero.)

Es muy importante saber que cualquier programa puede realizarse de formas diferentes, la escogida para solucionar este problema no es precisamente la más obvia y fue así para que el lector practique realizar este mismo programa de una manera diferente.

Ejercicio No 3:

Hacer un programa que reciba un número y determine si es un número perfecto. Un número es perfecto cuando la suma de sus divisores menores a él da como resultado el mismo número. Ej: 28 es un número perfecto pues sus divisores son: 14, 7, 4, 2, 1 y la suma de estos números da 28.

Análisis: Al igual que en el ejercicio 2, tenemos que hallar los divisores de un número, con una diferencia: ahora esos divisores debemos sumarlos, y al final comparar dicha suma con el número que introdujo el usuario, si los números son iguales imprimir que el número es “Perfecto” o lo contrario de no serlo. Algo a tener muy en cuenta es que la definición de numero perfecto nos dice: “La suma de sus divisores menores a él” ósea que en la suma no podemos incluir el numero al que le estamos sacando los divisores.

El código: Bastara con realizar unas pequeñas modificaciones al código del programa del ejercicio número 2:

```
1  (define (Divs A Cont Suma)
2    (if (= Cont 1)
3      (if (= A Suma)
4        (begin
5          (display Suma)
6          (display" Es un numero perfecto") )
7        (begin
8          (display A)
9          (display " No es un numero perfecto") )
10      )
11     (begin
12       (if (integer? (/ A Cont))
13         (Divs A (- Cont 1) (+ Suma (/A Cont)))
14         (Divs A (- Cont 1) Suma)
15        ) ) ) )
16 (define (Perfecto? X)
```

```
17      (Divs X X0) )
18  (Perfecto?28)
```

Análisis del código: Solo unas modificaciones con respecto a nuestro programa para calcular los divisores de un número.

En la línea uno, la definición de nuestra función principal, agregamos un parámetro llamado "Suma" que nos almacenara la suma de los divisores.

En la línea 3, cuando el caso base se cumpla, comparamos a Suma con A, de ser iguales significa que el número evaluado es un perfecto (se imprime en las líneas 5 y 6), de no ser iguales se imprime que no lo es (líneas 8 y 9).

Para la línea 13, cuando la división de A entre Cont sea un número entero se hace el llamado recursivo sumando el resultado de dicha división a Suma. Si el resultado de la división no da un entero se hace el llamado recursivo disminuyendo a Cont en 1.

Por último la línea 17 de la función Perfecto?, necesaria para hacer el programa más "estético", en el llamado a la función Divs el parámetro Suma se inicializa en 0, esto es porque aquí es donde se van sumando los divisores, si llamáramos la función con Suma diferente de 0 el resultado sería erróneo en todas las ocasiones.

Ejercicio No 4:

Se pide hacer un programa en DrRacket tal que, presente lo siguiente en la pantalla:

- Para el primer renglón solo se pide imprimir una letra "A" y luego se debe saltar al nuevo renglón, sin espacios en blanco.
- Para el segundo renglón se debe colocar un espacio en blanco, luego imprimir una letra "A" y después se debe saltar al nuevo renglón.
- Para el tercer renglón se debe colocar dos espacios en blanco, luego imprimir una letra "A" y después se debe saltar al nuevo renglón.
- Para el tercer renglón se debe colocar tres espacios en blanco, luego imprimir una letra "A" y después se debe saltar al nuevo renglón.

Así sucesivamente hasta 15 veces. El número de espacios en blanco en cada fila va aumentando así:

Ciclos	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Espacios en blanco	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Desarrollo:

- Función (Espacios): Utilizada para desplegar los espacios en blanco que se requieren para cada nuevo renglón
- a: parámetro que controla el fin de los llamados recursivos

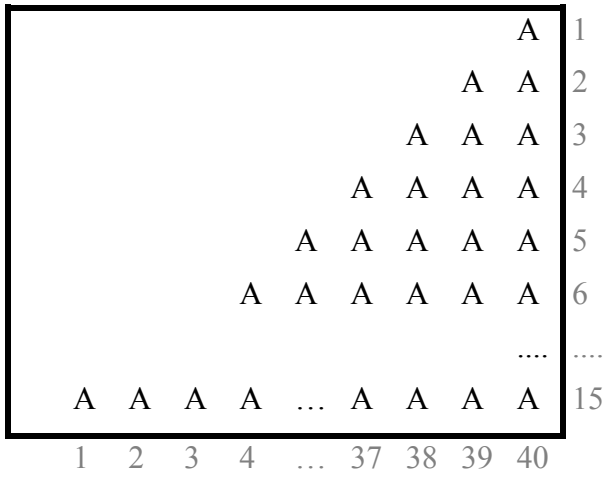
```
;Usando el lenguaje Muy Grande
(define (Espacios a)
  (if (> a 1)
    (begin
      (display" ")
      (Espacios (- a 1 ))
     )   ) )
```

- Función (Principal): Utilizada para contar los renglones de impresión con el parámetro a y en cada llamado recursivo de la función, hacer la impresión de los espacios en blanco y la letra “A”, con un salto de línea
- a: parámetro que cambia de uno en uno hasta llegar a quince, que son los renglones que se van a imprimir.

```
(define (Principal a)
  (if (<= a 15)
    (begin
      (Espacios a)
      (display"A")
      (newline)
      (Principal  (+ a 1) )
     )
   )
)
(Principal 1)
```

Ejercicio No 5:

Se pide hacer un programa en DrRacket tal que, que presente lo siguiente en la pantalla.



ANÁLISIS: De lo pedido se observa:

- Que para el primer renglón se deben desplegar treinta y nueve espacios en blanco, una letra “A” y saltar al nuevo renglón.

- Para el segundo renglón se debe desplegar treinta y ocho espacios en blanco, una letra “A” y después se debe saltar al nuevo renglón.
- Para el tercer renglón se debe desplegar treinta y siete espacios en blanco, una letra “A” y después se debe saltar al nuevo renglón.

El número de espacios en blanco en cada fila va aumentando así:

Ciclos	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Espacios en blanco	39	38	37	36	35	34	32	31	30	29	28	27	26	25	24
Cantidad de letras “A”	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

La variable que controla los ciclos, de acuerdo a los espacios en blanco se comporta como (– 40 ciclos); entonces en cada renglón, se colocará el número de espacios correspondientes, luego el número de letras “A” que es igual a la variable que controla los ciclos y por último el salto a nueva línea.

Lo anterior indica que debo construir las siguientes funciones:

Una función que despliegue los espacios en blanco

Una función que controle el despliegue de las letras “A”

Una función que controle el renglón a imprimir que para el caso son quince(15).

Desarrollo:

Función (espacios): Utilizada para desplegar los espacios en blanco que se requierenpara cada nuevo renglón

n: parámetro que controla el fin de los llamados recursivos

```
(define (Espacios n)
  (if (>= n 1)
    (begin
      (display " ")
      (Espacios (- n 1))
     )
   )
 )
```

Función (Cantidad_de_A): Utilizada para desplegar la letra “A” el número de veces que se requieran para cada nuevo renglón

n: parámetro que controla el fin de los llamados recursivos

```
(define (Cantidad_de_A n)
  (if (>= n 1)
```

```
    (begin
      (display "A")
      (Cantidad_de_A (- n 1))
        )
      )
  )
```

Función (Principal): Utilizada para contar los renglones de impresión con el parámetro a y en cada llamado recursivo de la función, hacer la impresión de los espacios en blanco y la letra “A”, con un salto de línea

n: parámetro que cambia de uno(1) en uno(1) hasta llegar a quince(15), que son los renglones que se van a imprimir.

```
(define (Principal n)
  (if (<= n 15)
    (begin
      (Espacios (- 40 n) )
      (Cantidad_de_A (- n 1) )
      (newline)
      (Principal (+ n 1) )
     )
   )
 )
```

Finalmente, el código quedaría así:

```
(define (Espacios n)
  (if (>= n 1)
    (begin
      (display " ")
      (Espacios (- n 1))
     )
   )
)
(define (Cantidad_de_A n)
  (if (>= n 1)
    (begin
      (display "A")
      (Cantidad_de_A (- n 1))
     )
   )
)
(define (Principal n)
```

```
18    (if (<= n 15)
19      (begin
20        (Espacios (- 40 n) )
21        (Cantidad_de_A (- n 1) )
22        (newline)
23        (Principal (+ n 1) )
24       )
25     )
26  )
27   (Principal 1)
```

6.4 ACTIVIDADES

1. Construir una función que reciba como parámetro un número N, y calcule la suma de todos los enteros menores que el número recibido
2. Escriba un programa que calcule la suma de los números que existen entre dos números dados. Debe incluir ambos números
3. Escriba una función que calcule cuantos números naturales hay entre 2 números dados, incluyéndolos
4. Hacer un programa en DrRacket tal que, dado un número entero, indique si es primo (Un número es primo si y solo si es divisible exactamente por si mismo y por la unidad). Utilice la función del ejercicio 2, que calcula la cantidad de divisores de un número.
5. Se pide hacer un programa en DrRacket tal que, presente lo siguiente en la pantalla.

Col/fila	1	2	3	4	5	6	7	8	9	10	11	12	13
1	P	P	P	P	P	P	P	P	P	P	P	P	P
2		P	P	P	P	P	P	P	P	P	P	P	
3			P	P	P	P	P	P	P	P	P		
4				P	P	P	P	P	P	P			
5					P	P	P	P	P				
6						P	P	P					
7							P						

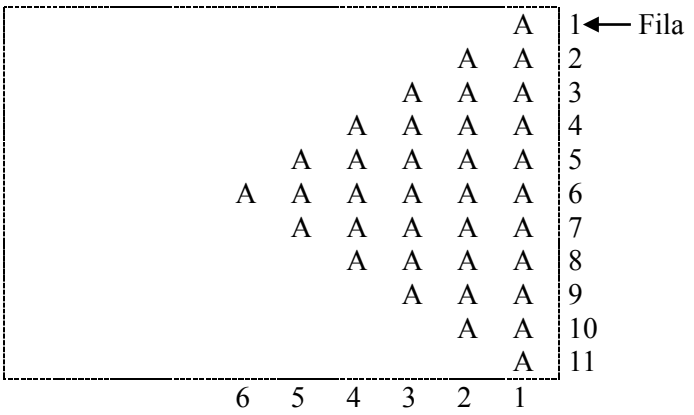
6. Se pide hacer un programa en DrRacket tal que, presente lo siguiente en la pantalla.

1	A																	A
2		B															B	
3			A													A		
4				B											B			
5					A									A				
6						B							B					
7							A					A						
8								B			B							
9									A	A								
10										B								

7. Se pide hacer un programa en DrRacket tal que, presente lo siguiente en la pantalla.

Col	1	2	3	4	5	6	7	8	9	10	11	12	13
Fila													
1	P	P	P	P	P	P	P	P	P	P	P	P	P
2		P	P	P	P	P	P	P	P	P	P	P	
3			P	P	P	P	P	P	P	P	P		
4				P	P	P	P	P	P	P			
5					P	P	P	P	P				
6						P	P	P					
7							P						
9						P	P	P					
10					P	P	P	P	P				
11				P	P	P	P	P	P	P			
12			P	P	P	P	P	P	P	P	P		
13		P	P	P	P	P	P	P	P	P	P	P	
14	P	P	P	P	P	P	P	P	P	P	P	P	P

8. Se pide hacer un programa en DrRacket tal que, que presente lo siguiente en la pantalla.



9. Se pide hacer un programa en DrRacket tal que, que presente lo siguiente en la pantalla

													Filas
	A											A	1
	A	A									A	A	2
	A	A	A							A	A	A	3
	A	A	A	A					A	A	A	A	4
	A	A	A	A	A				A	A	A	A	5
	A	A	A	A	A	A		A	A	A	A	A	6
	A	A	A	A	A				A	A	A	A	7
	A	A	A	A						A	A	A	8
	A	A	A								A	A	9
	A	A										A	10
	A											A	11
Columnas	28	29	30	31	32	33	34	35	36	37	38	39	40	

10. Se cuenta que al inventor del juego de ajedrez el Rey que le mando a hacer el juego le pregunto a él, como quería que le pagara, a lo que contesto que ubicará un grano de trigo en el primer cuadro del tablero de ajedrez, en el segundo el doble del primero, en el tercero el doble del segundo y así sucesivamente hasta completar los 64 cuadros del tablero. Se pide hacer un programa en DrRacket tal que, imprima la cantidad de granos de trigo que debió colocar el REY en cada cuadro del tablero para pagarle al inventor; así:

    PANTALLA

Cuadro	No granos de trigo
1	1
2	2
3	4
4	8
5	16
6	32
.....	
64	# # # # # #

    Condiciones: Solo podrá utilizar un máximo de diez funciones (display)

11. Calcular el factorial de un número.

    (*Factorial n)*=1*2*3*4*….*n

    Ej. (*Factorial* 0) = 1

    (Factorial 1) = 1

    (*Factorial* 2) = 1*2 = 2

    (*Factorial* 4) = 1*2*3*4 = 24

12. Implementar la serie (Fibonacci n): 0, 1, 1, 2, 3, 5, 8, 13, 21…

Ejemplos:

(Fibonacci 0)=0

(Fibonacci 4)=3

(Fibonacci 6)=8

13. Escribir un programa que pida un número y saque por pantalla su tabla de multiplicar hasta el 10
14. Escribir un programa para jugar a adivinar un número entre 1 y 10 (generado al azar por el ordenador) hasta acertarlo.

7

CARACTERES, COMENTARIOS Y DOCUMENTACIÓN

En DrRacket los caracteres son un tipo de dato más, que cuenta con una cantidad considerable de funciones para ser operados.

En este capítulo vamos a tratar la mayor parte de estas funciones, además iremos un poco más allá de los caracteres y veremos los comentarios y la documentación de un programa, además de la documentación guiada por recetas.

7.1 CARACTERES

Los caracteres son un tipo de dato. Los caracteres se utilizan para mostrarlos en pantalla y asi comunicarnos con el usuario. No debemos confundir un carácter con una cadena, aunque estos estén estrechamente ligados. Una cadena es un grupo de caracteres, por ejemplo: "Programar" es una cadena y está compuesta por un grupo de caracteres que son: {P, r, o, g, r, a, m, a, r}, pero no solo las letras del alfabeto son caracteres también existen otro tipo llamados "Caracteres especiales".

Sintaxis:

Los caracteres en DrRacket tiene una sintaxis especial, se escriben usando la notación #\<carácter>.

Caracteres especiales: Son especiales en dos sentidos: Porque no están contenidos en el alfabeto, y porque los caracteres se representan con una única letra, los especiales en cambio, se representan con palabras completas.

Ejemplos:

```
#\space      ; Imprime un espacio en blanco
#\newline    ; Imprime un salto de línea
```

Operaciones con caracteres: DrRacket nos provee de varias funciones para operar caracteres, veamos el uso de algunas de ellas:

`char?:` Es una función que determina si un dato es un carácter.

Sintaxis: `(char? X)`

Ejemplos:

- `(char? #\'a)` -> #t Devuelve verdadero porque a es un carácter.
- `(char? #\a)` -> #f Devuelve falso porque 'a es un símbolo no un caracter
- `(char? #\"Arena")` -> #f Devuelve falso porque "Arena" es una cadena no un caracter
- `(char? #\space)` -> #t Devuelve verdadero porque space es un caracter

Existen también funciones para encontrar si un carácter va antes o después de otro carácter en el código ASCII, o si son el mismo. Estas funciones son:

```
(char=? ch1 ch2)     ; Es ch1 el mismo caracter que ch2?
(char<? ch1 ch2)     ; ch1 está antes que ch2 en el alfabeto?
(char>? ch1 ch2)     ; ch1 está después que ch2 en el alfabeto?
(char<=? ch1 ch2)    ; ch1 está antes que ch2 en el alfabeto o son los mismos?
(char>=? ch1 ch2)    ; ch1 está después que ch2 en el alfabeto o son los mismos?
```

–ci, se usa para un caso insensible, es decir que no importará si hay mayúsculas o minúsculas, Racket las tomará como iguales.

Sintaxis:

```
(char-ci=? ch1 ch2)  ; Lo mismo que char=? Pero insensible
(char-ci<? ch1 ch2)  ; Lo mismo que char<? Pero insensible
(char-ci>? ch1 ch2)  ; Lo mismo que char>? Pero insensible
(char-ci<=? ch1 ch2) ; Lo mismo que char<=? Pero insensible
(char-ci>=? ch1 ch2) ; Lo mismo que char>=? Pero insensible
```

Ejemplos:

- `(char=? #\s #\S)` -> #f Compara basándose en el código ASCII, si la s minúscula está antes que la s mayúscula.
- `(char-ci=? #\s #\S)` -> #t Devuelve verdadero porque –ci hace que no se distinga la mayúscula de la minúscula por lo tanto Racket las evalúa como iguales.
- `(char<? #\a #\b)` -> #t Devuelve verdadero pues la “a” va antes que la “b” en el código ASCII.
- `(char<? #\a #\A)` -> #f Devuelve falso pues las mayúsculas van antes que las minúsculas en el código ASCII.

Racket también provee funciones para encontrar que tipo de caracter se está evaluando. Puede ser un caracter alfabético, numérico, espacio en blanco, mayúscula o minúscula, respectivamente como se muestra a continuación:

Sintaxis:

```
(char-alphabetic? ch)
(char-numeric? ch)
(char-whitespace? ch)
(char-upper-case? ch)
(char-lower-case? ch)
```

Ejemplos:

- `(char-alphabetic? #\a)` `-> #t`
- `(char-numeric? #\a)` `-> #f`
- `(char-numeric? #\2)` `-> #t`
- `(char-whitespace? #\space)` `-> #t`
- `(char-upper-case? #\A)` `-> #t`
- `(char-lower-case? #\A)` `-> #f`

También hay funciones para convertir caracteres a enteros en el conjunto de datos del código ASCII y viceversa. Estas son:

Sintaxis:

```
(char->integerch)
(integer->char n)
```

Ejemplos:

- `(char->integer #\3)` `-> 51`
- `(char->integer #\a)` `-> 97`
- `(integer->char 97)` `-> #\a`
- `(char->integer #\@)` `-> 64`
- `(integer->char 56)` `-> #\8`
- `(integer->char (char->integer #\a))-> #\a`

Em Racket podemos usar dos procedimentos para convertir caracteres de minúsculas a mayúsculas y viceversa. Ellos son:

Sintaxis

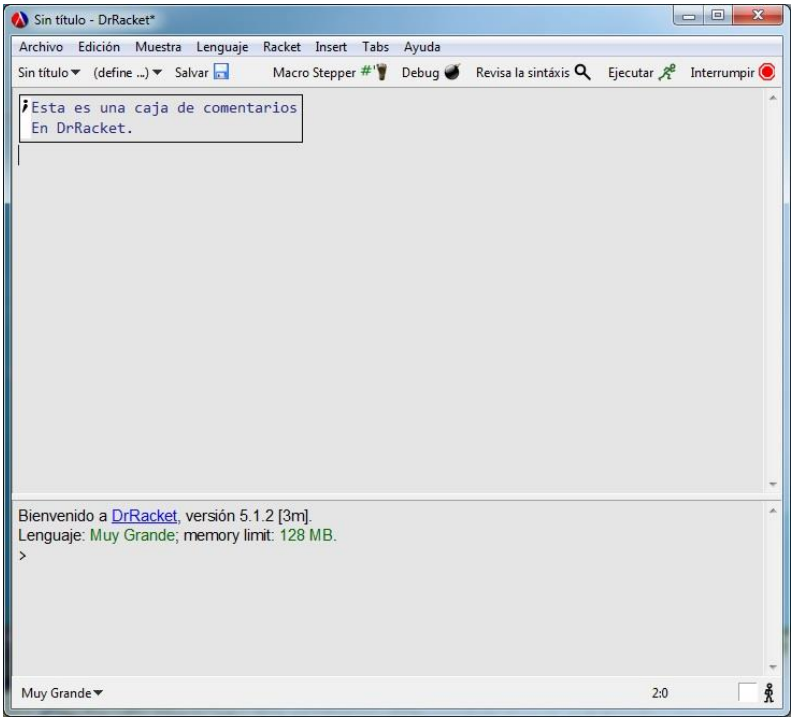
```
(char-upcasech)
(char-downcasech)
```

Ejemplos:

- `(char-upcase #\a)` `-> #\A`
- `(char-downcase #\B)` `-> #\b`
- `(char-upcase #\b)` `-> #\B`
- `(char-upcase (char-downcase #\B))` `-> #\B`

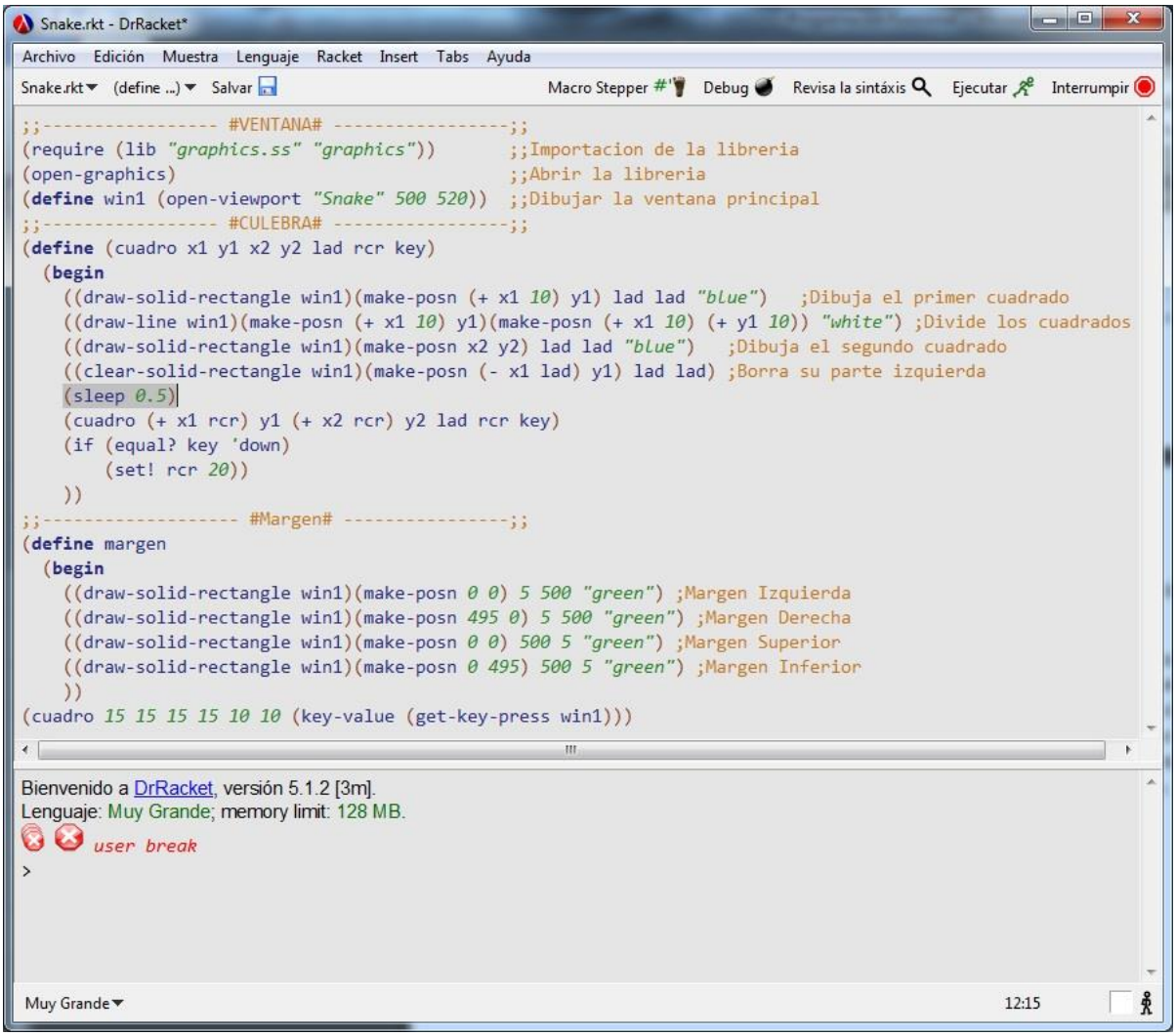
7.2 COMENTARIOS

Los comentarios, como habíamos visto en el capítulo 2, se declaran usando el carácter punto y coma y terminan en el siguiente salto de línea, pero también tenemos las "Cajas de comentarios": un recuadro en los que (sin importar la cantidad de saltos de línea o el texto que incluyamos)todo lo que este dentro del será considerado comentario, veamos una imagen:



Para usar una caja de comentarios basta con ir al menú "Insert" y luego dar clic en la opción "Insertar caja de comentarios".

Comentar un programa es incluso tan importante como el correcto funcionamiento del mismo, puesto que para diseñar grandes programas es necesario más de un programador y la mejor forma de trabajar en equipo es comentando el programa, igualmente si es un programa que desarrollas tu solo es normal que luego de un tiempo leas el código fuente y no tengas una idea clara de que hace cada sección de código, por eso si comentas el programa te estas ayudando a ti mismo y a tus colaboradores a mantener el código, corregir errores y optimizarlo.



Con comentarios también podemos dividir el código en secciones, de esta forma nos facilitaremos mucho el trabajo a la hora de corregir, revisar y/o optimizar.

La documentación de un programa es un tema muy extenso que en algunos casos esta guiado por normas estrictas que pueden variar dependiendo de tu grupo de trabajo o la empresa para la que escribas código, pero en general siempre es necesario especificar el objetivo de cada función, dividir el código en secciones con bloques de comentarios y añadir el "Contrato" o la documentación completa. La cual veremos a continuación:

7.3 Documentación

La documentación de un programa va desde comentar sus líneas de código hasta escribir el manual de usuario, en nuestro caso, veremos a grandes rasgos dos elementos importantes en la documentación: El diseño guiado por recetas o Contrato y El uso de los comentarios, los demás elementos que también forman parte de la documentación no serán nombrados dado que este libro solo es una guía de introducción al mundo de la programación.

Diseño guiado por contrato:

Necesitamos determinar qué es lo relevante en la declaración de un problema y que podemos ignorar. Necesitamos entender que recibe el programa, que produce y como se relacionan sus entradas y salidas. Debemos conocer, o encontrar, si Racket provee ciertas operaciones básicas que nos sirvan en nuestro programa. Si no, debemos desarrollar programas auxiliares que implementen estas operaciones. Finalmente, una vez tengamos un programa, debemos comprobar si el programa hace lo que debe hacer. Esto puede revelar los diferentes tipos de errores.

```
; Contrato: AreaDelAnillo: numero numero  ->  número
; Propósito: Calcular el área de un anillo.
; Ejemplo: (AreaDelAnillo 5 3) debe producir 50.24
; Parámetros:
;         Exterior: Es el valor del radio exterior del anillo
;         Interior: Es el valor del radio interior del anillo
; Definición: [refines theheader]
(define (AreaDelAnillo exterior interior)
  (- (AreaDelDisco exterior)
     (AreaDelDisco interior)))
; Pruebas:
(AreaDelAnillo5 3)
; Valor esperado:50.24
```

Entendiendo el propósito del programa:

La meta de diseñar un programa es crear un mecanismo que reciba y produzca datos. Debemos comenzar cada programa, dándole un nombre significativo y determinando que clase de información recibe y produce. A esto se le llama CONTRATO.

Aquí es como nosotros escribimos un contrato para AreaDelAnillo, uno de nuestros programas:

; AreaDelAnillo: numero numero -> numero

El contrato consiste de dos partes. La primera, a la izquierda de los dos puntos, declara el nombre del programa. La segunda, a la derecha de los dos puntos, especifica qué clase de datos recibe el programa y que debe producir o devolver; las entradas son separadas de la salida por una flecha ->.

Después usando el contrato debemos formular una DECLARACION DE PROPÓSITO corta, esto es, un breve comentario de qué debe calcular el programa. Para la mayoría de nuestros programas, una o dos líneas serán suficiente; Para programas más grandes se debe adicionar más información.

En el EJEMPLO podemos determinar qué respuesta debe entregar el programa y así observar si está funcionando bien.

En los PARAMETROS se explicará para que sirva cada una de los parámetros que llegan a la función.

Después se escribe la Función y por último se deben hacer pruebas para comprobar la buena funcionalidad del programa.

Notas: Si la declaración del problema provee una fórmula matemática, el número de variables diferentes en la fórmula sugiere cuantos parámetros recibe el programa.

Hacer ejemplos, antes de escribir el programa, ayuda en muchas formas. Primero, es la única forma segura de descubrir errores lógicos. Segundo, los ejemplos nos fuerzan a pensar sobre el proceso computacional, el cual, para los casos complicados que encontraremos después, es crítico para el desarrollo del cuerpo de la función. Finalmente, los ejemplos ilustran lo que dice la declaración del propósito. Los futuros lectores de los programas, como profesores, colegas o compradores, apreciaran todas estas ilustraciones de conceptos.

Para formular el cuerpo de un programa se requiere tener conocimiento acerca del área, también conocida como dominio, para el cual el programa estará desarrollado.

Ya que los programadores no pueden conocer todos los dominios de la computación, deben estar preparados para entender los lenguajes de programación muy bien para entonces poder discutir los problemas con expertos del dominio.

Documentación de código:

Documentar el código de un programa es añadir información suficiente como para explicar lo que hace, paso por paso, de forma que no solo la maquina sepa que hacer, sino que además los humanos entiendan que está haciendo y por qué.

Porque entre lo que tiene que hacer un programa y cómo lo hace hay una distancia impresionante: todas las horas que el programador ha dedicado a pergeñar una solución y escribirla en el lenguaje que corresponda para que el ordenador la ejecute ciegamente.

Documentar un programa no es sólo un acto de buen hacer del programador por aquello de dejar la obra rematada. Es además una necesidad que sólo se aprecia en su debida magnitud cuando hay errores que reparar o hay que extender el programa con nuevas capacidades o adaptarlo a un nuevo escenario. Hay dos reglas que no se deben olvidar nunca:

Todos los programas tienen errores y descubrirlos sólo es cuestión de tiempo y de que el programa tenga éxito y se utilice frecuentemente.

Todos los programas sufren modificaciones a lo largo de su vida, al menos todos aquellos que tienen éxito.

Por una u otra razón, todo programa que tenga éxito será modificado en el futuro, bien por el programador original, bien por otro programador que le sustituya. Pensando en esta revisión de código es por lo que es importante que el programa se entienda: para poder repararlo y modificarlo.

Que hay que documentar?

Fácil: Todo lo que no es evidente. Que hace esta función? Para que este parámetro? Y esta librería, que contiene? Se está usando un algoritmo especifico? Que se debería mejorar… si hubiera tiempo?

Esos son el tipo de preguntas que uno se hace al leer el código de un programa, por tanto son las partes a comentar del mismo.

Como hago un comentario? Facil, ya se ha explicado muchas veces en este libro: Usando el carácter “Punto y coma” (;). Así de ahora en adelante todo programa por pequeño que sea, escrito por nosotros debe llevar comentarios explicando almenas que hace y como lo hace.

7.4 EJEMPLOS CON CARACTERES Y DOCUMENTACIÓN

Ejercicio No 1:

Recibir un caracter que representa una nota. Si es E (mayúscula) ó e (minúscula) imprimir "Excelente", si es B ó b imprimir "Bueno", si es A o a imprimir "Aceptable", si es D o d imprimir "Deficiente" y si es I o i imprimir "Insuficiente".

Análisis: Necesitamos un programa que reciba una letra e indiferentemente de si esta es mayúscula o minúscula el programa debe devolver un resultado. Vamos a usar dos funciones principalmente, "cond" para nuestro condicional y "char-ci=?" para verificar que letra es, sin diferenciar mayúsculas o minúsculas

Código:

```
(define (Notas X) ;El argumento X debe ser reemplazado con un carácter
  (cond ;Usamos cond ya que es idóneo para este tipo de ejercicios
    [(char-ci=? X #\e) (display "Excelente")]
    [(char-ci=? X #\s) (display "Sobresaliente")]
    [(char-ci=? X #\a) (display "Aceptable")]
    [(char-ci=? X #\i) (display "Insuficiente")]
    [(char-ci=? X #\d) (display "Deficiente")]
    (else (display "No es una nota valida."));Si se introduce un caracter inesperado.
    )
  )
(Notas (read))
```

Notas: El usuario debe introducir un carácter con la sintaxis correcta, (#\) si se envía solamente la letra tecleada arrojara un error.

Ejercicio No 2:

Imprimir los caracteres que equivalen a los números hasta el 1024(código ASCII).

Análisis: Igual que en el ejemplo anterior usaremos principalmente dos funciones, if para nuestra condición y ciclo recursivo, y integer-char para convertir valores enteros a caracteres.

```
(define(CodASCII Cont)
  (if(= Cont 1024) ;Cont aumentara hasta 1024 y se dará el caso base
      (display (integer->char Cont)) ;(caso base)
      (begin ;Para todos los casos generales
        (display (integer->char Cont)) ;Imprimir Cont como un caracter
        (display " ") ;Imprimir un espacio para separar los caracteres
        (CodASCII(+ 1 Cont)) ;Llamado recursivo, aumenta Cont en 1
```

```
        )
      )
  )
(CodASCII 1)
```

Ejercicio No 3:

Realice el programa propuesto y documente el código siguiendo el diseño guiado por contrato:

Escribir un programa que pida un número y saque por pantalla su tabla de multiplicar hasta el 10.

Código:

```
; Contrato: TablaMultiplicar: numero numero -> numero cadena numero cadena numero...
; Propósito: Visualizar determinada tabla de multiplicar.
; Ejemplo: (TablaMultiplicar 8 0) debe producir la tabla de multiplicar del número 8 hasta el 10, con el formato: 8*0 = 0 ... 8*10 = 80
; Parámetros:
;        a: Indica que numero multiplicar
;        b: Actúa como contador y siempre debe iniciar en 0
; Definición:
(define (TablaMultiplicar a b)
  (if (= b 10)
      (begin
        (display a)
        (display "*")
        (display b)
        (display " = ")
        (display (* a b))
        )
      (begin
        (display a)
        (display "*")
        (display b)
        (display " = ")
        (display (* a b))(newline)
        (TablaMultiplicar a (+ 1 b))
        )
      )
  )
; Pruebas:
(TablaMultiplicar (read) 0)
; Valor esperado: Debe devolver la tabla de multiplicar del número ingresado
;                  por teclado hasta multiplicarse por 10, con el siguiente formato:
;                a*0 = 0
```

```
;                   ...
;                   a*10 = a*10
```

7.1 ACTIVIDADES

1. Hacer una función que reciba un parámetro y devuelva verdadero si es un carácter.
2. Hacer una función que reciba un parámetro. Si el parámetro es un caracter alfabético, determinar si está en minúscula. El programa debe pasarlo a mayúscula y retornarlo. Hacer lo mismo en caso contrario
3. Hacer una función EsVocal que reciba un parámetro y devuelva verdadero si el parámetro es una vocal y falso de lo contrario. No importa si la vocal es minúscula o mayúscula.
4. Hacer una función que reciba un caracter e imprima "Es una vocal" si el caracter es una vocal o devuelva un mensaje de error en caso contrario. Usar la función anterior.
5. Hacer una función que compare 2 caracteres y devuelva si son iguales, o si el primero va antes que el segundo en el código ASCII o si el segundo va antes que el primero
6. Hacer una función que reciba un parámetro y retorne "es alfanumérico" si el parámetro es un carácter alfanumérico o "es numérico" si es numérico o "es un espacio en blanco" si es un espacio en blanco y lo mismo si es mayúscula y minúscula.
7. Hacer una función que reciba un parámetro, si el parámetro es un caracter devolver el número que corresponda en la tabla del código ASCII y si es un número devolver el caracter que corresponda en la tabla. **Nota:** la función ***(number? n)***, retorna verdadero si ***n*** es un número y falso de lo contrario
8. Hacer una función que reciba un parámetro. Si el parámetro es un caracter alfabético, determinar si está en minúscula y pasarlo a mayúscula y retornar este valor. Hacer lo mismo en caso contrario.
9. Escribir un programa que pida una palabra acabada en un punto y cuente las letras que contiene
10. Escribir un programa que pida una palabra y cuente el número de vocales y consonantes que contiene
11. Documente con el Diseño guiado por contrato los programas anteriores
12. Agregue comentarios a todos los programas que hizo y que realizara.

8

CADENAS/STRINGS

Las cadenas están estrechamente ligadas a los caracteres. Recordemos: #\a es un carácter pero "Adelante" es una cadena. Con lo que podemos ver el concepto de cadena: Una cadena es un conjunto de caracteres.

En Racket una cadena se representa entre comillas dobles, ejemplo "Esto es una cadena", podemos referirnos a cada uno de los caracteres que componen la cadena con su índice, por ejemplo: La cadena "Esto" contiene cuatro caracteres y sus índices son de la siguiente manera:

Cadena	E	S	T	O
índice	0	1	2	3

Como sabemos Racket tiene funciones primitivas para manejar cada tipo de dato, las cadenas no son la excepción

8.1 CREACIÓN Y MODIFICACIÓN DE CADENAS

Para crear una cadena basta con declararla, esto se hace tecleando comillas dobles, todo lo que este dentro de ellas se considera una cadena, si no hay nada escrito entre las comillas se considera una cadena vacía.

Racket proporciona la función **string?** para comprobar si un dato es una cadena, su sintaxis es la siguiente:

Sintaxis:

```
(string? n)
```

Ejemplos:

```
(string? "Casa"); -> #t, Devuelve verdadero porque es una cadena de caracteres
(string? 'bobo);  -> #f, Porque 'bobo es un símbolo
(string? 5);      -> #f, Porque 5 es un número
(string? "");     -> #t, Es verdadero pues es una cadena vacía
```

Racket también proporciona varias funciones para crear y manipular cadenas.

Las cadenas se pueden crear con procedimientos make-string y string.

Sintaxis:

```
(make-string n)
(make-string n ch)
(string ch1 ch2 ...)
```

Ejemplos:

- `(make-string 3); -> "\u0000\u0000\u0000"`

  Crea una cadena de 3 posiciones y la llena con ceros.

- `(make-string 3 #\a); ->  "aaa"`

  Crea una cadena de 3 posiciones y la llena con el caracter #\a.

- `(string #\R #\a #\c #\k #\e #\t); -> "Racket"`

  Crea una cadena con los caracteres dados.

La longitud de una cadena es el número de caracteres que la componen. Por ejemplo la longitud de la cadena vacía es 0, la longitud de la cadena "Racket" es 6.

Racket proporciona una función para calcular la longitud de una cadena: s`tring-length`

Sintaxis:

El procedimiento **`(string-length cadena)`**, toma una cadena como argumento y retorna un número entero que corresponde al tamaño de la cadena. Por ejemplo:

Ejemplos:

- `(string-length "Scheme");    -> 6`
- `(string-length "");          -> 0`

Las cadenas pueden también ser concatenadas (unidas) usando la función (string-append cadena1 cadena2….), que toma varias cadenas y retorna una cadena con la unión de todas las cadenas.

Ejemplos:

- `(string-append "futbol" " villa" "deporte"); -> "futbol villadeporte"`
- `(string-append); -> ""  Devuelve la cadena vacía`

Cadenas modificables:

Una cadena es modificable cuando por medio de operaciones esta cadena puede cambiar en su contenido. **Ejemplo:** la cadena "Maria" si fuese modificable podría cambiarse por "Mario".

Los procedimientos **`(string ch1 ch2 ...)`**, **`(make-string n ch)`** y **`(string-append cadena1 cadena2…)`** pueden usarse para crear cadenas modificables. Pero una cadena creada mediante comillas dobles no es modificable

Racket provee de una función que puede modificar una cadena: **`string-set!`**

Sintaxis:

(string-set! cadena *n caracter*) ;Donde cadena es la cadena a modificar, n es el índice del carácter a modificar y carácter es el carácter por el cual será reemplazado.

Ejemplo:

```
(string-set! (string #\e #\j #\e) 2 #\a)
```

Para obtener caracteres de una cadena se usa el procedimiento **(string-ref cadena n),** que toma a "cadena" y devuelve el carácter que se encuentre en el índice "n". Recordar que el primer carácter tiene el índice 0.

Ejemplo:

```
(string-ref "América" 0); -> #\A, retorna el caracter de la cadena en la posición 0.
```

La función **(substring cadena *comienzo final*)**, en donde ***comienzo*** y ***final*** son números y retorna una subcadena que empieza en el caracter de la posición ***comienzo*** y terminando en el caracter de la posición ***final*** restándole 1**.**

Ejemplo:

```
(substring "futbol" 1 3); -> "ut", Empieza en el caracter 1 y termina en el
caracter 2
```

La funcion **(string-set! cadena *n caracter*)** almacena el ***caracter*** en el elemento ***n*** de la **cadena**. En otras palabras reemplaza el carácter de ***cadena*** cuyo índice es ***n*** con ***carácter***.

Ejemplos:

```
(define str (string #\f #\u #\t #\b #\o #\l)) ; "futbol"
(string-set! str 2 #\d); reemplaza d en el caracter no. 2 de cadena
```

8.2 OPERACIONES DE COMPARACIÓN EN CADENAS

Racket también proporciona una variedad de funciones o predicados para determinar la el orden de dos cadenas.

Si dos cadenas son de diferente longitud y una cadena es prefijo de la otra cadena, la cadena más corta es considerada lexicográficamente menor que la otra.

Ejemplo: la cadena "carro" es prefijo de la cadena "carrotanque" por lo tanto la cadena "carro" es considerada menor que la otra cadena.

Sintaxis

```
(string=? cadena1 cadena2)  ; Es cadena1 igual que cadena2?
(string<? cadena1 cadena2)  ; Es cadena1 lexicográficamente menor que cadena2?
(string>? cadena1 cadena2)  ; Es cadena1 lexicográficamente mayor que cadena2?
(string<=? cadena1 cadena2) ; Es cadena1 lexicográficamente menor o igual cadena2?
(string>=? cadena1 cadena2) ; Es cadena1 lexicográficamente mayor o igual cadena2?
```

```
(string-ci=? cadena1 cadena2)      ; Igual que string=? Pero insensible
(string-ci<? cadena1 cadena2)      ; Igual que string<? Pero insensible
(string-ci>? cadena1 cadena2)      ; Igual que string>? Pero insensible
(string-ci<=? cadena1 cadena2)     ; Igual que string<=? Pero insensible
(string-ci>=? cadena1 cadena2)     ; Igual que string>=? Pero insensible
```

Ejemplos:

```
(string=? “hola” “HOLA”);     -> #f Se consideran como cadenas diferentes
(string-ci=? “hola”  “HOLA”); -> #t El procedimiento es insensible
(string<? “bar" “futbol");    -> #t bar está antes en el alfabeto que futbol
(string<? "futbolbar" "futbol");   -> #f futbol es prefijo y menor que futbolbar
```

8.3 EJEMPLOS CON CADENAS

Ejercicio Nº 1

Escribir un programa que sirva para generar códigos de usuario por el procedimiento siguiente: Tiene que leer el nombre y los dos apellidos de una persona y devolver un código de usuario formado por las tres primeras letras del primer apellido, las tres primeras letras del segundo apellido y las tres primeras letras del nombre. Por ejemplo, si la cadena de entrada es: “BALVIN SANCHEZ ANA” debe devolver “BALSANANA”.

Análisis: Se nos pide tomar las tres primeras letras de los dos apellidos y del primer nombre de una persona para generar un código. A primera vista podría parecer un ejercicio complicado, pero realmente es muy sencillo, solo debemos hacer un correcto uso de la función substring y estará listo.

Código

```
(define (Codigo nom ap1 ap2)
  (display (substring nom 0 3))
  (display (substring ap1 0 3))
  (display (substring ap2 0 3))
  )

(Codigo (read)(read)(read))
```

Recordemos que existen muchas formas de escribir un mismo programa, en nuestro caso hemos asumido que el usuario ya sabe que debe ingresar el texto en el orden: primer apellido, segundo apellido y nombre, además que debe hacerlo encerrando entre comillas cada uno de estos y separándolos con espacios. Para efectos de un programa cuya finalidad sea comercial o de presentación a un público desconocedor del lenguaje Racket debemos agregar unos cuantos display para indicar al usuario como ingresar el texto, o llegar modificar la programación de las funciones para hacerlo algo más presentable, incluso este programa puede desarrollarse sin el uso

de la función substring.

Ejercicio Nº 2

Cree un programa que reciba una cadena y devuelva otra sin las vocales.

Nota: Antes de leer el código, escribe el programa tú mismo, recuerda que un mismo programa puede escribirse de muchas formas distintas, luego de escribir tu propio código, compáralo con este y observa que cosas se pueden mejorar de tu código y el del libro.

```
(define (Cadena read) ;Lee y almacena nuestra Cadena.
  Read
 )
(define (Tamaño read) ;Define y almacena el tamaño de nuestra Cadena.
  (string-length (Cadena read))
 )
(define (Sinvocls CAD LENGTH POS)
;/Devuelve Cadena sin vocales, CAD  referencia a la cadena para evaluar,
LENGTH a su tamaño
  (if (= LENGTH POS)
    (display "")
    (begin
      (if(or(char-ci=? (string-ref(Cadena CAD) POS) #\a)
;Evalúa si el carácter de Cadena en la posición POS es igual a una de las vocales.
             (char-ci=? (string-ref(Cadena CAD) POS) #\e)
             (char-ci=? (string-ref(Cadena CAD) POS) #\i)
             (char-ci=? (string-ref(Cadena CAD) POS) #\o)
             (char-ci=? (string-ref(Cadena CAD) POS) #\u))
             (display "") ;De cumplirse, devolver vacío.
             (display (string-ref(Cadena CAD) POS))
            )
        (Sinvocls CAD LENGTH (+ POS 1));Llamado recursivo.
       )
     )
  )
(define(Principal usrdef) ;Esta función simplifica el llamado.
  (Sinvocls(Cadena usrdef) (Tamaño usrdef) 0)
  )
(Principal (read))
```

8.4 Actividades

1. Hacer una función que reciba una cadena y devolverla pero invertida.
2. Haga una función que reciba dos cadenas y devuelva otra cadena con la unión de ellas. No usar string-append
3. Escribir un programa que sirva para generar códigos de usuario por el procedimiento

siguiente: Tiene que leer el nombre y los dos apellidos de una persona y devolver un código de usuario formado por las tres primeras letras del primer apellido, las tres primeras letras del segundo apellido y las tres primeras letras del nombre. Por ejemplo, si la cadena de entrada es: "BALVIN SANCHEZ ANA" debe devolver "BALSANANA" (No usar substring).

4. Hacer una función que reciba una cadena y devuelva cuantas vocales tiene
5. Hacer una función que reciba una cadena e indique si es palíndroma. Una cadena de caracteres es palíndroma si se lee igual al derecho que al revés. Ejemplo: "ojo", "abcdedcba", "a", "dabalearrozalazorraelabad" "atar a la rata". Nota: Para frases palíndromas, recuerde que debe introducir toda la frase entre comillas dobles, además el programa primero debe eliminar los caracteres extraños y de espacio.
6. Hacer una función que reciba dos cadenas de caracteres e indique si la segunda está incluida en la primera (es decir si es subcadena) Ejemplo: La cadena "Mar", está incluida en la cadena "Maracaibo"
7. Recibir 2 cadenas y determinar cual es menor. En caso contrario imprimir "Son iguales". Ej: "Bucaramanga" es menor que "Cali", por el orden del alfabeto.
8. Recibir una cadena. Si tiene longitud mayor de 10 caracteres imprimir "Es muy larga esta cadena" de lo contrario imprimir "La cadena es pequeña".
9. Recibir una cadena de longitud mayor a 10 caracteres y devolver una subcadena con los caracteres de la cadena recibida del caracter 2 al 8.
10. Recibir una cadena e imprimir el primer y el último caracter.
11. Recibir una cadena, cambiar el último carácter por el caracter #\x y devolver la cadena modificada.
12. Hacer una función que reciba un entero N y devuelva una cadena de longitud N, leyendo por pantalla cada uno de los N caracteres de la cadena.
13. Recibir una cadena y contar cuantos espacios en blanco tiene.
14. Hacer una función que reciba una cadena de caracteres y cuente cuantas vocales tiene, cuantas consonantes, cuantos espacios, cuantos caracteres extraños y cuantos números, y principalmente cuente de cuantas palabras está compuesta la cadena. Nota: El usuario debe introducir una oración a manera de cadena (encerrándola toda entre comillas).
15. Hacer una función que reciba una cadena de caracteres y devuelva otra cadena solamente con sus consonantes.
16. Hacer un programa que reciba una cadena. El programa debe tomar cada uno de los caracteres que componen la cadena y reproducirlos tantas veces como caracteres tenga la cadena. Ejemplo: la cadena "Perro" tiene 5 caracteres por lo tanto el programa deberá devolver: "PPPPPeeeeerrrrrrrrrrooooo"

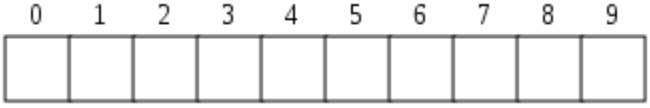
9

VECTORES

- El tipo de dato vector, es conocido como un tipo de dato compuesto de uno o más tipos de dato básico o primitivo o de otros tipos de datos compuestos, inclusive vectores mismos.
- En programación, una matriz o vector es una zona de almacenamiento continuo que contiene una serie de elementos del mismo tipo, los elementos del vector.
- Desde el punto de vista lógico un vector se puede ver como un conjunto de elementos ordenados en fila (o filas y columnas si tuviera dos dimensiones).

Los elementos de un vector pueden ser a su vez vectores, lo que nos permite hablar de vectores multidimensionales. Estas estructuras de datos son adecuadas para situaciones en las que el acceso a los datos se realice de forma aleatoria e impredecible. Por el contrario, si los elementos pueden estar ordenados y se va a utilizar acceso secuencial sería más adecuado utilizar una lista, ya que esta estructura puede cambiar de tamaño fácilmente durante la ejecución de un programa

Como vemos, existe variedad de formas para conceptualizar un vector en programación, pero, para darlo a entender de su forma puramente práctica, un vector (en Racket) es un procedimiento que nos permite almacenar en sí mismo determinada cantidad de datos, (como enteros, cadenas, caracteres, vectores etc.) esta cantidad es establecida por el usuario, además otras funciones o procedimientos pueden hacer referencia a un elemento del vector mediante índices, siendo el primer elemento dentro del vector de índice 0, y el ultimo de índice n. (Muy similar a los índices para cadenas que vimos en el capítulo anterior).



Vector unidimensional de 10 elementos.

9.1 SINTAXIS DE VECTORES

```
(vector (+ 2 2) 1 #\q 'Wooow "¡Hola!" )
```

Este vector, tiene 5 elementos, el primero de ellos (cuyo índice es 0) es la función (+ 2 2), el segundo es el numero entero 1, el tercero es el carácter #\q, el cuarto es el símbolo 'Wooow, y el quinto es la cadena "¡Hola!".

La función vector, crea un vector únicamente con los elementos que se le indiquen en el momento de declarar dicha función. Adicionalmente podemos modificar dichos elementos o hacer referencia a ellos, también existen funciones que nos permiten crear un vector vacío con n posiciones, o de n posiciones con un mismo dato dentro de todas ellas. En Racket a diferencia de la mayoría de lenguajes de programación podemos almacenar absolutamente cualquier tipo de dato en nuestros vectores.

9.2 CREACIÓN, MODIFICACIÓN Y MANIPULACIÓN DE VECTORES.

Make-vector, es una función para crear un vector con 1 o dos parámetros, el primero siempre

debe ser un valor numérico que indica la cantidad de posiciones y el segundo (opcional) es un tipo de dato cualquiera para llenar todas las posiciones del vector.

Sintaxis:

```
(make-vector 6 #\w);-> #(#\w #\w #\w #\w #\w #\w)
(make-vector 5)     ;-> #(0 0 0 0 0 0)
```

Vector, crea un vector con una cantidad de posiciones como parámetros incluyamos en la función. Tiene n parámetros y pueden ser de cualquier tipo

Sintaxis:

```
(vector "X" #\a 'b 2 3 1 1 (+ 2 1));->#("X" #\a b 2 3 1 1 3)
```

Vector-ref, devuelve el dato ubicado en la posición referenciada del vector. Tiene dos parámetros, el primero es el vector o el nombre de la función donde está definido, el segundo es el índice del vector donde se encuentra el dato que queremos obtener.

Sintaxis:

```
(define vect (vector 12 "X" "Hola" 23))
(vector-ref vect 2) ;-> "Hola"
```

```
(vector-ref (vector "x" 1 23) 2) ;-> 23
```

Vector-set!, procedimiento por el cual se toma un elemento de un vector y se modifica cambiándolo por otro. Tiene 3 argumentos, el primero es el vector o nombre de la función donde está definido, el segundo es el índice del dato a modificar y el tercero es el nuevo dato para esa posición.

Sintaxis:

```
;Se define una constante "vec2" para guardar el vector:
(define vec2 (vector 1 2 3))
;Ingresa 876 en la posición 0 del vector "vec2":
(vector-set! vec2 0 876)
vec2                    ;-> #(876 2 3)
```

Vector->list, Convierte un vector en una lista, tiene un solo parámetro y es el vector a convertir.

Sintaxis:

```
(define vec (vector 12 "1" "Hola" "amigos"))
(vector->listvec) ;-> (12 "1" "Hola" "amigos")
```

Vector-fill!, Ingresa un dato dado en todas las posiciones del vector, tiene dos parámetros, el nombre del vector y el dato a introducir en las posiciones de este.
Sintaxis:

```
(define vec (vector 12 "1" "Hola" "amigos") )
(vector-fill! vec "hola")  ;-> #("hola" "hola" "hola" "hola")
(vector-fill! vec 56)            ;-> #(56 56 56 56)
```

Vector-length, Esta función cuenta la cantidad de posiciones de un vector, tiene un solo argumento y es el vector a evaluar.

Sintaxis:

```
(define vec (vector 12 "1" "Hola" "amigos") )
(vector-lengthvec);-> 4
```

9.3 EJEMPLOS DE PROGRAMAS CON VECTORES

Ejercicio Nº 1

Dado un vector con diferentes valores imprimir dichos valores del vector leyéndolos uno por uno de manera recursiva:

Análisis:

Debemos definir un vector en una función y llenar este con determinado número de datos, luego, usando otra función debemos leer la primera posición del vector (el número que indique esta posición debe ser un argumento en la función) y mostrarla por pantalla,en el llamado recursivo debemos aumentar en uno la posición a leer del vector hasta haber leído y visualizado todas las posiciones del vector, pero antes se debe conocer el número de posiciones en el vector, de otra forma no podríamos finalizar correctamente la recursión, este número también debe ser un argumento.

Codigo:

```
(define Vect (vector "Esto" " " "Es" " " "Un" " " "Vector" " " "Leído" " " "Recursivamente" "."))
(define (DisplayVectLength Pos)
  (if (= (- Length 1) Pos)
;El tamaño del vector es 12, pero su últimaposición es 11 (porque empiezan a contar en 0) por lo tanto Length se debe restar en 1
      (display(vector-refVect Pos))
      (begin
        (display (vector-refVect Pos))
        (display VectLength (+ 1 Pos))
        )
     )
 )
(DisplayVect (vector-lengthVect) 0)
```

Ejercicio Nº 2

Hacer un programa que reciba un numero entero y cree un vector de tamaño n, donde cada posición del vector se llene del índice 0 al 9 con los números del 1 al 10, en caso que el vector ser más grande de diez posiciones, las restantes se deben llenar con las letras del abecedario desde la "a" hasta la "z" y en caso tal que el número de posiciones del vector sea aún mayor, las posiciones siguientes deben llenarse con la misma secuencia de números (1 al 10) y seguido las letras del abecedario. Nota: Sin importar el tamaño del vector este debe llenarse con lo indicado anteriormente.

Análisis:

Si bien a primera vista puede parecer que este es un programa complicado de resolver, a medida que pensemos en cómo resolverlo (y lo hagamos) nos daremos cuenta que no lo es tanto. Se nos pide crear un vector de un tamaño definido por el usuario, para esto podemos definir una función y crear el vector con make-vector y (read) como parámetro, esta función también nos servirá como referencia para las posiciones del vector (restándole 1, ya que empiezan a contar en 0) y para el numero de llamados recursivos a realizar. Las posiciones del vector se deben llenar con números del 1 al 10 y luego con las letras del abecedario, hasta ese punto no tenemos nada complicado pero según se nos pide aunque el tamaño del vector sea mayor a 37 (a los que corresponden las primeras diez posiciones para los números y las 26 siguientes para las letras) se debe seguir introduciendo la misma secuencia que en principio hasta llenar por completo el vector.

¿Cómo podemos lograr esto? Como siempre, existen varias formas de hacer un programa y quizá tú idees una mejor a la que yo usare y explicare a continuación:

Código:

```
1  (define Vec(make-vector (read)))
2  (define (LlenarVec Length Pos LlenarNum LlenarChar n)
3    (if (= Length Pos)
4      (display Vec)
5      (begin
6        (if (<= n 10)
7          (begin
8            (vector-set! Vec Pos LlenarNum)
9            (LlenarVecLength (+ 1 Pos) (+1 LlenarNum) 97 (+1 n)) )
10         (begin
11           (vector-set! Vec Pos (integer->charLlenarChar))
12           (if (= n 36)
13             (LlenarVecLength (+1 Pos)1 (+ 1 LlenarChar) 1)
14             (LlenarVecLength (+1 Pos) 1 (+ 1 LlenarChar) (+ 1 n))
15             ) )
16           )
17      ) )
```

```
18    )
19
20  (LlenarVec (vector-length Vec) 0 1 97 1)
```

Análisis del código:

Realicemos un breve análisis: La línea 1 contiene la función **Vec**, que es nuestro vector. En la línea dos inicia la función principal "**LlenarVec**", con varios argumentos, ***Length***: Es el tamaño del vector. ***Pos***: Es la posición a evaluar del vector. ***LlenarNum***: Para fines de simplicidad en el código he usado dos parámetros para llenar con ellos el vector, este corresponde a los números. ***LlenarChar***: corresponde al número en código ASCII equivalente a la letra "a", (97) e ira aumentando hasta llegar a "z". ***n***: Con n nos encargaremos de que cuando se termine de llenar el vector con los números se continúe con las letras y cuando estas terminen sigan los números y así sucesivamente.

Línea 6: Si n es menor o igual a 10 significa que aún no se ha llenado el vector con los números del 1 al 10, cuando n sea mayor que 10 se iniciara el llenado del vector con las letras del abecedario.

Línea 8: Es la función para llenar el vector en la posición actual.

Línea 9: Es el llamado recursivo, en donde, la posición se aumenta en 1, el numero con el que llenar el vector también aumenta en uno (recordemos que debe llegar hasta 10) y n que nos indica si llenar con números o con caracteres aumenta en uno. LlenarChar se envía iniciado en 97 para el momento en que n sea 11 la posición 10 del vector se llene con el carácter equivalente al número 97.

Línea 11: se encarga de llenar el vector con los caracteres.

Linea 12: Se encarga de evaluar a n para saber si es igual a 36, de serlo significa que ya se ha llenado el vector con los números del 1 al 10 y con todas las letras del abecedario, así que en el llamado siguiente n se reiniciara en 1 (para volver a empezar con los numeros), en el caso que n sea diferente de 36 se continuara el llamado recursivo aumentando a n en 1. En los dos casos **LlenarNum** se *establece*como 1 para queel vector se llene correctamente al iniciar de nuevo con los números.

9.4 Actividades

10. Hacer un programa que reciba un vector, y muestre el último dato del vector
11. Hacer un programa que reciba un valor entero N y devuelva un vector de tamaño N, donde cada posición del vector contiene el valor -1.
12. Dado un vector con diferentes valores imprimir dichos valores del vector leyéndolos uno por uno de manera recursiva.
13. Dado un número n, crear un vector de tamaño n y luego ingresar en el vector los números del 1 al n, e imprimir el vector. Ej: Dado el numero=5 ingresar en el vector (vector 1 2 3 4 5).
14. Dado un vector con solo números, realizar la suma de sus elementos.
15. Llenar un vector V de 10 elementos con los cuadrados de los 10 primeros elementos. Ej: (vector 1 4 9 16 25 36 49 64 81 100)

16. Dado un vector con n números. Calcular el promedio de los elementos del vector.
17. Hacer un programa que reciba un entero N y devuelva un vector de tamaño N, con varios valores leídos por teclado.
18. Dado un vector V de enteros y un número X, devolver el valor que corresponde al número de veces que está X en el vector.
19. Dado un vector, hacer un programa que invierta sus datos y devuelva el vector invertido.
20. Hacer un programa que reciba un vector de enteros y un número X, la función debe borrar el número si lo encuentra y debe hacer SHIFT-LEFT (mover a la izquierda) todos los elementos siguientes y dejando el valor -1 en la última posición. Considerar que todos los números son diferentes.
21. Hacer un programa reciba un vector de enteros y un numero X, busque el número X en el vector y devuelva la posición donde se encuentra la primera vez ese número en el vector. En caso de no estar debe devolver #f.
22. Hacer un programa que reciba un vector de enteros y devuelva el número mayor y la posición en el vector donde se encuentre. Los números pueden ser positivos o negativos.
23. Dados dos vectores con elementos ya incluidos, devolver un tercer vector con la concatenación de los dos vectores dados.
24. Definir un vector de 9 posiciones de forma global (Recordar: Declaración de constantes). Hacer un programa e ingresar los números del 1 al 9 en el vector de forma aleatoria y mostrarlo en pantalla. Nota: los números se pueden repetir.
25. Hacer el mismo ejercicio anterior pero los números no se pueden repetir.
26. Hacer un programa que reciba un vector de enteros e indique si está ordenado ascendentemente.
27. Hacer un programa que reciba un vector de enteros y lo ordene ascendentemente.
28. Hacer un programa que reciba un vector de números. Se deben modificar cada uno de los datos, multiplicando su valor por 100. Ej: si el vector es: #(3 5 8 2), debe quedar así: #(300 500 800 200)

10

LISTAS, PARES

"Para el uso de estas funciones será necesario en algunos casos usar el lenguaje R5RS."

10.1 PARES

Un par es una estructura de datos con dos campos llamados cabeza y cola. Los pares son creados con el procedimiento **cons**. Se puede ingresar cualquier tipo de dato a los pares.

Veamos las funciones que se usan para operar con pares:

Cons: crea un par, tiene dos argumentos. El primero: la cabeza y el segundo: la cola

Sintaxis:

```
(cons a b) ;-> (a . b) La cabeza es a y la cola es b.
```

Car: Extrae la cabeza de un par, tiene un argumento que es el par a evaluar.

Sintaxis:

```
(define par (cons 1 "hola"))
(display (car par)) ;->1
```

Cdr: Extrae la cola de un par, tiene un argumento que es el par a evaluar

Sintaxis:

```
(define par (cons 1 "hola"))
(display (cdr par)) ;->"hola"
```

Un par se diferencia de una lista en que la cola puede contener otros pares. (Pares anidados) Veamos:

```
(define par (cons 1 (cons 2 (cons 3 (cons 4 5)))))
(display (cdr par)) ;-> (2 3 4 . 5)
(display (car par)) ;-> 1
```

Pair?: Procedimiento que determina si un objeto es un par o no, tiene un solo argumento y es el objeto a evaluar.

Sintaxis:

```
(pair? (cons 1 "hola")) ;-> #t
```

Los pares son usados principalmente para representar las colas y las listas. Las listas se diferencian de los pares en que estos solo admiten 2 elementos, mientras que las listas admiten más de 2 elementos y terminan en un elemento vacío *(empty)*, los pares no. Las listas se crean con el procedimiento **list**. Se puede decir que una lista es un par pero un par NO es una lista.

10.2 Listas

Una lista es un tipo de dato en Racket con cierta similitud a un vector, pero las listas se diferencian principalmente de los vectores en que estas son dinámicas, es decir se les puede añadir información, mientras que los vectores son estáticos. En una lista podemos incluir n número de datos y estos pueden ser de cualquier tipo. Una lista siempre terminara en un espacio vacio *((empty), aunque este no sea visible).*

Veamos las funciones que se usan para operar las listas:

List: crea una lista, tiene tantos argumentos como elementos queramos incluir en nuestra lista.

Sintaxis:

```
(list 5 4 6 "Hola" 'bl #\a (vector 4 5 6 1) (list 2 3 4) (+ 2 2))
;-> (5 4 6 "Hola" 'bl #\a #(4 5 6 1) (2 3 4) 4)
```

Car, Cdr: Tienen el mismo uso que en los pares, sirven para extraer la cabeza y la cola (respectivamente) de una lista.

Sintaxis:

```
(car (list 5 4 6));-> 5
(cdr (list 5 4 6));-> (4 6)
```

Null?: Es el procedimiento que determina si una lista esta vacía. Tiene un solo argumento y es la lista a evaluar. Devuelve un booleano.

Sintaxis:

```
(define Lista (list 5 4 6))
(null? Lista) ;-> #f
```

```
(define Lista (list))
(null? Lista) ;-> #t
```

Append: Es el procedimiento que nos permite añadir elementos a una lista.

Sintaxis:

```
(define (AdicionarNumeros Lista n)
  (if (<= n 10)
```

```
      (AdicionarNumeros (append Lista (list n)) (+ n 1))
      Lista
  )
)
(AdicionarNumeros (list) 1) ; -> (1 2 3 4 5 6 7 8 9 10)
```

```
(define Lista (list 1 2 3))
(append Lista 4) ; -> (1 2 3 . 4)
```

```
(define Lista (list 1 2 3))
(append Lista (list 4)) ; -> (1 2 3 4)
```

List?: Este procedimiento determina si un elemento es una lista, tiene un solo argumento y es la lista a evaluar.

Sintaxis:

```
(define Lista (list 1 2 3))
(list? Lista) ; -> #t
```

Length: Es el procedimiento que determina el tamaño de una lista. Un solo argumento, es la lista a evaluar.

Sintaxis:

```
(define Lista (list 1 2 3))
(length Lista) ; -> 3
```

Reverse: Invierte una lista. Tiene un solo argumento, es la lista a invertir.

Sintaxis:

```
(define Lista (list 1 2 3))
(reverse Lista) ; -> (3 2 1)
```

List-tail: Esta función devuelve una subcola de una lista, tiene dos argumentos el primero es la lista y el segundo es el índice desde donde sacar la cola.

Nota: Los índices en las listas empiezan en 0 y terminan en un elemento vacío.

Sintaxis:

```
(define Lista (list "Lunes" "Martes" "Miércoles" "Jueves" "Viernes"))
(reverse Lista 2) ;-> ("Miércoles" "Jueves" "Viernes")
```

10.3 Ejemplos de programas con listas y pares

Ejercicio 1:

Hacer una función que reciba una lista e imprima uno por uno todos sus valores. Usar la función null? Para saber si llegó al final de la lista.

Código:

```
(define (Printlist Lista)
  (if (not (null? Lista))
    (begin
      (display (car Lista))
      (newline)
      (Printlist (cdr Lista))
     )
   )
 )
(Printlist (list 5 4 3 2 0 12 93))
```

Análisis:

Recordemos que la única forma de recorrer una lista o un par es usando las funciones ***car*** y ***cdr***, es por eso que en el llamado recursivo de la función el argumento *Lista* es remplazado por la cola de la misma, para que no se tome el primer elemento, pues este ya se visualizó, y quede en la cabeza el siguiente.

Ejercicio 2:

Hacer una función que devuelva una lista con los datos que el usuario digite por teclado. La entrada de datos termina cuando el usuario entre el número -1.

Código:

```
(define (Agregar lista leer)
  (if (equal? leer -1)
      (display lista)
      (Agregar (append lista (list leer)) (read))
      )
  )
(Agregar (list) (read))
```

Análisis: Después de leer el código nos podemos dar cuenta que la solución es mucho más sencilla de lo que tal vez imaginamos. Ha de tener muy en cuenta el uso de ***equal?*** pues este es necesario para poder introducir cualquier tipo de dato a la lista. (Recordemos que "=" únicamente evalúa datos de tipo numérico.) También en el llamado recursivo a la función remplazar *leer* con *(read)* es de vital importancia, pues de no hacerlo *leer* siempre tomara el valor que ingresamos en el primer *read* sin importar cuantos o cuales mas ingresemos.

10.4 ACTIVIDADES

1. Haga una función que reciba una lista y retorne la suma de sus elementos (usar recursión)
2. Hacer una función que reciba una lista y cuente de forma recursiva cuantos elementos hay en la lista.
3. Hacer una función que reciba una lista y un dato a buscar. Se debe devolver la posición en la que se encuentra el dato la primera vez en la lista o -1 sino existe.
4. Hacer una función que reciba una lista de enteros y devuelva el mayor valor contenido en la lista o -1 si está vacía.
5. Hacer una función que reciba una lista de enteros e indique si está ordenada ascendentemente.
6. Hacer una función que reciba una lista y devuelva otra con los datos de la primera invertida. Nota: No se puede usar otra estructura de datos como un vector.
7. Haga una función que reciba una lista de números y un número. Si este número no se encuentra en la lista, ingresarlo a ella en la parte final.
8. Haga una función que reciba una lista de boléanos y retorne true si al menos un valor es true ó devuelva false si ningún valor es true.
9. Hacer una función que reciba una lista de nombres (cadenas de caracteres) y devuelva la cadena de mayor longitud.
10. Hacer una función que reciba números enteros y devuelva una lista de listas. La lista debe contener 4 listas. En la primera están los números impares negativos, en la segunda los pares negativos, en la tercera los impares positivos y en la última los pares positivos. Recuerde que la lista vacía se crea con (list).

11

ESTRUCTURAS DE DATOS

En programación, una estructura de datos es una forma de organizar un conjunto de datos con el objetivo de facilitar su manipulación. Una estructura de datos define la organización e interrelación de estos y un conjunto de operaciones que se pueden realizar sobre ellos.

Las estructuras son tipos de datos que tienen varias piezas de información. Por ejemplo una estructura llama *Persona* puede tener información importante como el *Nombre* de la persona, la *Cedula,* la *Dirección, Teléfono,* etc. Similar a este ejemplo, las estructuras se pueden utilizar en cualquier momento donde un dato contenga otros datos. En rezumen, varias piezas de datos que componen un solo dato pueden componer una estructura. Las estructuras se pueden usar para ingresar registros en archivos o en bases de datos.

Una forma de pensar gráficamente una estructura es una caja con registros y campos. (Celdas, filas y columnas):

Alumnos

Nombre	Apellido	Contacto
Carolina	Gomez	carogomez@clubpr.com
Diego	Osorio	dieosorio@clubpr.com
Michael	Muñoz	micmunoz@clubpr.com
Alejandro	Cardona	alecardona@clubpr.com
Cristian	Nieto	nietocrist@clubpr.com
Sebastián	Molina	molinasebas@clubpr.com

11.1 LAS ESTRUCTURAS DE DATOS EN DRRACKET

En DrRacket definir una estructura es sencillo, se logra mediante el procedimiento: ***define-struct***, dicho procedimiento tiene un uso casi idéntico al de definir una funcion usando *define,* primero debemos declarar un nombre para nuestra estructura y luego entre paréntesis sus elementos. Veamos:

```
(define-struct Alumnos (Nombre Apellido Contacto))
```

Se pueden utilizar dos funciones (respectivamente) para ingresar datos o leerlos de una estructura:

make-NombreEstructura: Ingresa datos a una estructura.

Sintaxis:

```
(make-Alumnos "Jose" "Valdes" "josevaldes@clubpr.com")
```

Dado que la estructura *Alumnos* tiene tres elementos, a la hora de agregar una nueva entrada esta debe contener la misma cantidad de elementos, estos elementos deben ser tipos de dato validos en Racket. En nuestro ejemplo se creó un nuevo registro para la estructura *Alumnos* con los valores: "Jose" en el campo *Nombre,* "Valdes" en el campo *Apellido* y josevaldes@clubpr.com en el campo *Contacto*.

NombreEstructura-Elemento: Lee datos de una estructura.

Sintaxis:

```
(Alumnos-Nombre (make-Alumnos "Juan" "Soto" "juansoto@clubpr.com")
;-> Juan
```

Leer un dato de una estructura puede resultar, en principio, confuso, así que veámoslo de otra forma: En este caso la estructura Alumnos solo contendrá dos *registros*, y estos son *A1* y *A2,* si queremos leer un dato específico debemos indicar el *registro* y el *campo* a leer.

```
(define-struct Alumnos (Nombre Apellido Contacto))
(define A1 (make-Alumnos "Pedro" "Vallejo" "pdrvllejo@clubpr.com")
(define A2 (make-Alumnos "Jose" "Cardona" "josecardona@clubpr.com")
(Alumnos-Apellido A2) ;-> Cardona
```

La función Alumnos-Apellido se definió automáticamente al crear la estructura Alumnos, de la misma forma se definieron las funciones: make-Alumnos, Alumnos?, Alumnos-Nombre, Alumnos-Contacto.

12

MODO GRAFICO

"Se entiende por modo gráfico, toda aquella interfaz en computación que involucre el uso de ventanas y ratón."

La gran mayoría de lenguajes de programación nos ofrece un "Modo grafico" o un conjunto de instrucciones y/o funciones que permiten usar ventanas, pixeles, multimedia y ratón, con el fin de desarrollar programas en modo gráfico. DrRacket no es la excepción y en este capítulo veremos el uso de las librerías Graphics e Image cuyas funciones nos permitiran realizar este tipo de programas.

12.1 INTRODUCCIÓN AL MODO GRAFICO

En la historia del software las aplicaciones han evolucionado gigantescamente, si miramos una o dos décadas atrás nos daremos cuenta que los entornos en los que se ejecutaban los programas eran muy limitados, pantallas de muy pocos pixeles, monocromáticas e interfaces de modo texto, con el paso de los años se fueron implementando nuevas tecnologías que permitieron tener pantallas cada vez más grandes, resoluciones mayores y el uso de color, que empezó con paletas de 8 colores y hoy día alcanzan más de 16 millones de colores o 32bits a lo que llamamos color verdadero.

Es innegable la tendencia al uso de las GUI (graphicuser interface), o interfaces graficas de usuario, cada programa que se desarrolla para el usuario final posee una interface gráfica y por este motivo todo programador debe saber desarrollarlas.

Racket provee de varias librerías para realizar gráficos, en este capítulo veremos la librería *Graphics* a grandes rasgos, mas sin embargo nombraremos las principales funciones para las demás librerías también muy usadas: *Image* y *Draw*, el uso de estas queda para investigación del lector.

Nota: Si queremos crear un ejecutable de nuestro programa, es necesario que usemos el lenguaje "Estudiante Avanzado", este será el lenguaje que usaremos para todos los ejemplos expuestos en este capítulo.

12.2 LA LIBRERÍA GRAPHICS

Para realizar un programa en modo grafico primero debemos definir con que librería de gráficos vamos a programar, en nuestro caso será "Graphics". Para indicárselo a Racket debemos agregar la siguiente línea de código al inicio de nuestro programa.

```
(require (lib "Graphics.ss""graphics")
(open-graphics)
```

La función *require* llama a la librería Graphics que contiene el conjunto de funciones necesarias para programar gráficos.

La función ***open-graphics*** inicializa la librería graphics, como sería normal la función ***close-graphics*** cierra la librería gráfica y con ella todas las ventanas y procesos que estemos realizando

con gráficos, para volver a usar gráficos debemos volver a usar la función *open-graphics*.

Como es lógico cualquier grafico debe ejecutarse en una ventana, por lo que lo primero que debemos hacer al programar gráficos es crear una ventana:

Open-viewport, es una función que crea una ventana, tiene tres argumentos, el primero debe ser una cadena y corresponde al nombre o título de la ventana, el segundo al ancho de la misma y el tercero a su altura.

```
(open-viewport "Ejemplo" 800 600)
```

Sleep, Esta función causa un retardo o espera de n segundos, tiene un solo argumento y es un valor numérico que define el número de segundos a esperar.

```
(sleep 3)
```

Close-viewport, Cierra una ventana de tipo viewport, tiene un solo argumento y es la ventana a cerrar.

```
(define Ventana (open-viewport "Mi Ventana" 800 600))
(close-viewport Ventana)
```

La forma de referirnos a una ubicación pixel a pixel en una ventana es mediante coordenadas x, y. Siendo la esquina superior izquierda el pixel (0,0) y las dichas posiciones se incrementan hacia la derecha y hacia abajo.

```
(struct posn (x y)) ;Es una posición dentro de la ventana.
```

Podemos colorear todo el contenido de una ventana mediante la función ***draw-viewport.*** Esta función tiene dos argumentos el primero es la ventana a colorear, y el segundo es el color a usar, este puede definirse en dos formatos: RGB o el predefinido por Racket en el que bastaría ingresar la cadena correspondiente al nombre del color en inglés:

```
((draw-viewportVentana) (make-rgb 0.7 0.1 1))
((draw-viewportVentana) "green")
```

Clear-viewport, limpia todo el contenido de la ventana, en otras palabras la deja de color blanco, solo requiere un argumento y es la ventana a *limpiar*.

```
((clear-viewportVentana))
```

Mediante la función ***draw-pixel,*** podemos pintar un pixel del color que deseemos. Esta función tiene tres argumentos, el primero es el nombre de la ventana, el segundo la posición del pixel a

colorear y el tercero el color a usar, igual que en la función anterior podemos usar el formato RGB o el definido por Racket.

```
((draw-pixel Ventana) (make-posn 200 202) "black")
((draw-pixel Ventana) (make-posn 200 203) (make-rgb 0.1 0.1 0.1))
```

Podemos borrar un pixel con la función ***clear-pixel,*** esta función tiene dos argumentos, el primero de ellos es la ventana y el segundo la posición.

```
((clear-pixel Ventana) (make-posn 200 202))
```

Flip-pixel, invierte el color del pixel en de la ventana en la posición dada:

```
((flip-pixel Ventana) (make-posn 200 202))
```

Podemos crear, borrar e invertir el color de líneas rectas mediante las siguientes tres funciones respectivamente:

```
((draw-line Ventana)(make-posn 60 50) (make-posn 400 200) "green")
;En donde la primera instrucción de posición corresponde al inicio de la
línea, y la segunda al final.
```

```
((clear-line Ventana)(make-posn 60 50) (make-posn 400 200))
```

```
((flip-line Ventana) (make-posn 60 50) (make-posn 400 200))
```

Podemos dibujar rectángulos no solidos con la función ***draw-rectangle***, esta tiene cuatro argumentos y son: la ventana, la posición, las dimensiones y el color. Con la función ***clear-rectangle***, podemos borrar un rectángulo no sólido, veamos la sintaxis de estas dos funciones respectivamente:

```
((draw-rectangle Ventana) (make-posn 100 100) 50 100 "blue")
```

```
((clear-rectangle Ventana) (make-posn 100 100) 50 100)
```

Con la función ***draw-solid-rectangle*** podemos dibujar un rectángulo sólido, tiene los mismos argumentos que la función anterior y podemos usar ***clear-solid-rectangle,*** para borrarlo. Veamos:

```
((draw-solid-rectangle Ventana) (make-posn 100 100) 50 100 "red")
```

```
((clear-solid-rectangle Ventana) (make-posn 100 100) 50 100)
```

Para finalizar con las figures geométricas tenemos la elipse, se puede crear, borrar e invertir su color con las siguientes funciones respectivamente:

```
((draw-ellipse Ventana) (make-posn 100 100) 300 100 "red")
;Dibuja una elipse
```

```
((clear-ellipse Ventana) (make-posn 100 100) 300 100)
;Borra una elipse
```

```
((flip-ellipse Ventana) (make-posn 100 100) 300 100 "red")
;Invierte el color de una elipse
```

Elipses solidas:

```
((draw-solid-ellipse Ventana) (make-posn 100 100) 300 100 "yellow"))
;Dibuja una elipse solida, de color Amarillo, recordemos que también se
 puede usar el formato RGB para definir el color.
```

```
((clear-solid-ellipse Ventana) (make-posn 100 100) 300 100)
;Borra una elipse solida
```

```
((flip-solid-ellipse Ventana) (make-posn 100 100) 300 100)
;Borra una elipse solida
```

Uso de cadenas en modo gráfico:

Con la función ***Draw-string***, podemos dibujar en la ventana que deseemos una cadena, esta función consta de cuatro parámetros, La ventana, la posición, la cadena y el color.

```
((draw-string Ventana) (make-posn 100 100) "DrRacket RULES!"
"red")
```

Al igual que en las demás funciones el color también puede ser definido en el formato RGB

```
((clear-string Ventana) (make-posn 100 100) "DrRacket RULES!")
;Borra una cadena
```

```
((flip-string Ventana) (make-posn 100 100) "DrRacket RULES!")
;Invierte el color de una cadena
```

Uso de imágenes:

Podemos importar imágenes dentro de nuestra ventana, con la función ***draw-pixmap***, veamos:

```
((draw-pixmap Ventana) "Img01.bmp" (make-posn 0 0))
```

Nota: Los formatos soportados son: gif, jpeg, png, xbm, xpm, bmp.

La función ***Copy-viewport*** copia todos los elementos gráficos de una ventana a otra, veamos:

```
(copy-viewport Ventana1 Ventana2)
```

Operaciones con el ratón:

Get-mouse-click, Esta función tiene un solo argumento y es la ventana en la cual leer la acción del mouse. Se espera a que el usuario de clic en la ventana y devuelve un *descriptor*.

Nota: Un DESCRIPTOR contiene la posición donde se dio clic, así como cual botón se oprimió.

```
(get-mouse-click Ventana)
```

Ready-mouse-click, Esta función recibe una ventana y no espera a que el usuario haga clic. Devuelve un descriptor de tipo true o false dependiendo de si se dio clic o no.

```
(ready-mouse-click Ventana)
```

Ready-mouse-release, Esta función recibe una ventana y retorna verdadero si se dejó de oprimir un botón del mouse o retorna falso en caso contrario

```
(ready-mouse-release Ventana)
```

Query-mouse-posn, Esta función recibe una ventana y retorna la POSICION del cursor dentro de la ventana o retorna falso si se encuentra fuera de ella.

```
(query-mouse-posn Ventana)
```

Mouse-click-posn, Esta función recibe un DESCRIPTOR y retorna la posición donde se dio clic dentro de la ventana.

```
(mouse-click-posn (get-mouse-click Ventana))
```

Left-mouse-click?, Esta función recibe un DESCRIPTOR y Retorna verdadero si se dio clic izquierdo y falso en caso contrario.

```
(left-mouse-click? (get-mouse-click Ventana))
```

Middle-mouse-click? Esta función recibe un DESCRIPTOR y Retorna verdadero si se dio clic del centro y falso en caso contrario.

```
(middle-mouse-click? (get-mouse-click Ventana))
```

Right-mouse-click? Esta función recibe un DESCRIPTOR y Retorna verdadero si se dio clic derecho y falso en caso contrario.

```
(right-mouse-click? (get-mouse-click Ventana))
```

Operaciones con el teclado:

Get-key-press, Esta función espera a que el usuario oprima una tecla y cuando sucede devuelve un descriptor.

```
(get-key-press Ventana)
```

Ready-key-press, Esta función no espera a que el usuario oprima una tecla y en caso de presionarse se devuelve un descriptor.

```
(ready-key-press Ventana)
```

Key-value, Esta función recibe un DESCRIPTOR y devuelve el carácter que corresponde a la tecla que fue presionada.

```
(key-value (get-key-press Ventana))
```

12.3 La librería Image

La librería "*image.ss"* al igual que graphics contiene un conjunto de funciones necesarias para realizar gráficos. Veremos solo las principales funciones ya que este capítulo se basa principalmente en el uso de la librería Graphics.

Siempre que deseemos usar una librería debemos *llamarla* haciendo uso de la función ***require:***

```
(require (lib "image.ss" "teachpack/htdp"))
```

Image?, Determina si un elemento es una imagen. Un argumento: El elemento a evaluar.

```
(image? (rectangle 100 10 'outline 'red))
```

Image-color?, Determina si un elemento es un color valido. Un argumento: El elemento a evaluar.

```
(image-color? (make-color 255 0 0))
```

Rectangle, Dibuja un rectángulo. Cuatro argumentos: Ancho, Alto, Modo: Puede ser *'solid* o *'outline,* Color.

```
(rectangle 40 40 'outline 'black)
```

Circle, Dibuja un circulo. Tres argumentos: Radio, Modo, Color.

```
(circle 30 "outline" 'green)
```

Ellipse, Dibuja una elipse. Cuatro argumentos: Ancho, Alto, Modo, Color.

```
(ellipse 50 150  "solid"  (make-color 255 0 0))
```

Triangle, Dibuja un triángulo. Tres argumentos: Lado, Modo, Color.

```
(triangle 100 "solid"  (make-color 0 255 0))
```

Star, Dibuja una estrella. Cinco argumentos: Numero de puntas, Radio exterior, Radio interior, Modo, Color.

```
(star 6 100 30 'solid 'red)
```

Regular-polygon, Dibuja un polígono. Cinco argumentos: Numero de lados, Tamaño de los lados, Modo, Color, Angulo.

```
(regular-polygon 5 200 'outline 'RED 0.95)
```

Line, Dibuja un alinea desde las coordenadas (0,0) hasta las dadas por el usuario. Tres argumentos: Coordenadas en X, Coordenadas en Y, Color.

```
(line 30 0 (make-color 0 0 255))
```

Add-line, Adiciona una línea a un elemento ya existente. Seis argumentos: Elemento existente (nombre de la función), Coordenada X en el elemento existente, Coordenada Y en el elemento existente, Coordenada final en X, Coordenada final en Y, Color.

```
(define Lin (line 50 0 'blue))
;definiremos a Lin como una línea
(add-line Lin 50 0 50 100 'red)
```

Text, Imprime una cadena de texto, Tres argumentos: Cadena a imprimir, Tamaño de letra, Color.

```
(text "Hola amigos como están" 50 'red)
```

Image-width, Obtiene el ancho de una imagen en pixeles. Un argumento: La imagen a evaluar.

```
(image-width (line 10 0 'red))
```

Image-height, Obtiene el alto de una imagen en pixeles. Un argumento: La imagen a evaluar.

```
(image-height (line 10 0 'red))
```

Overlay, Superpone la imagen n sobre la imagen n-1 y está sobre la imagen n-2, así como imágenes hallan. n Argumentos: Todos los argumentos pueden ser elementos gráficos, se superpondrán unos a otros dependiendo del orden, siendo el último argumento el que este superpuesto a los demás.

```
(define trazo (line 50 100 'black)) ;LINEA
(define equi (triangle 30 'solid 'blue) ) ;TRIANGULO EQUILATERO
(define texto (text cadena 20 'red) );TEXTO que usa la cadena anterior.
(overlay  texto trazo equi) ;Sobrepone el triángulo sobre la linea y
esta sobre el texto
```

Overlay/xy, Superpone una imagen a otra en una distancia indicada. Cuatro argumentos: Imagen a superponer, Coordenada en X, Coordenada en Y, Segunda imagen.

```
(define trazo (line 50 0 'red) )
(define cadena (string #\H #\o #\l #\a) )
(define texto (text cadena 20 'blue) )
(overlay/xy  texto 100 20  trazo)
```

12.4 EJEMPLOS DE PROGRAMAS EN MODO GRAFICO

En esta parte construiremos funciones que sirven como base para programas más completos, también programaremos un pequeño juego que quizá todos conocemos, el Ahorcado. Todo esto usando la librería graphics.

Programar gráficos no es diferente a lo que hemos programado anteriormente, solo combinaremos las funciones que ya conocíamos con las incluidas en la librería gráfica, usaremos los mismos conceptos anteriormente mencionados y lo único nuevo que debemos tener muy en cuenta son las coordenadas x, y, para ubicarnos en la ventana.

Ejercicio Nº 1:

Cree un programa en donde en una ventana de 800*600 se dibuje un pixel en la posición (0,50), luego debe dibujarse otro pixel del mismo color en la posición (1,50), (2,50), (3,50) hasta llegar a la posición (800,50). Use recursividad.

Código:

```
1  (require (lib "graphics.ss" "graphics"))(open-graphics) ;Llamado de librería
2  (define ventana1 (open-viewport "Linea de puntos" 800 600))
3  (define (linea x y) ;x, y, son coordenadas.
4    (if (= x 799) ;Caso base
5      ((draw-pixel ventana1)(make-posn 800 50) "green");Dibuja el ultimo pixel
6      (begin
7        ((draw-pixel ventana1)(make-posn x y) "green") ;Dibuja el pixel en la posición (x,y)
8        (sleep 0.001) ;Tiempo de espera para dibujar el siguiente pixel.
9        (linea (+ x 1) y)  ;Llamado recursivo
10      )      )  )
11 (linea 0 50)
```

Ejercicio Nº 2:

Realice un programa en el que: Se dibuje un cuadrado en pantalla y luego mediante el uso de las teclas de dirección este cuadro se mueva alrededor de la pantalla a antojo del usuario.

Análisis:

Para comenzar debemos dibujar un cuadrado en pantalla, este lo manipularemos con el teclado y

variaremos su posición, lo que nos dice que: El cuadrado debe estar definido en una función, la posición en el eje *x*, y en el eje *y,* deben ser argumentos de la función.

Código:

```
1  (require (lib "graphics.ss" "graphics"))
2  (open-graphics)
3  (define win1 (open-viewport "Cuadro" 800 600))
4  (define (cuadro x y lad)
5    (begin
6      ((draw-solid-rectangle win1) (make-posn x y) lad lad "blue")
7      ((clear-solid-rectangle win1)(make-posn (- x lad) y) lad lad)
8      ((clear-solid-rectangle win1)(make-posn (+ x lad) y) lad lad)
9      ((clear-solid-rectangle win1)(make-posn (- x lad)(- y lad))(* 3
   lad)lad)
10     ((clear-solid-rectangle win1)(make-posn x(+ y lad)) (* 3 lad) lad)
11    ))
12 (define (movimiento x1 y1 lad1 key)
13   (if (equal? key 'down)
14     (begin
15       (cuadro x1 y1 lad1)
16       (movimiento x1 (+ y1 5) lad1 (key-value (get-key-press win1))) )
17       (if (equal? key 'up)
18         (begin
19           (cuadro x1 y1 lad1)
20           (movimiento x1 (- y1 5) lad1 (key-value (get-key-press
   win1))) )
21           (if (equal? key 'right)
22             (begin
23               (cuadro x1 y1 lad1)
24               (movimiento (+ x1 5) y1 lad1 (key-value (get-key-press
   win1))) )
25               (if (equal? key 'left)
26                 (begin
27                   (cuadro x1 y1 lad1)
28                   (movimiento(- x1 5) y1 lad1(key-value(get-key-press
   win1))) )
29                   (movimiento x1 y1 lad1 (key-value (get-key-press
   win1)))
30                 )))))
31 (movimiento 60 60 100 (key-value (get-key-press win1)))
```

Analicemos el código: La función *cuadro*, dibuja un cuadrado y a la vez borra su rededor, esto es necesario porque una vez se presione una tecla de dirección el cuadro ya dibujado no se moverá, solo se dibujara uno nuevo en la posición siguiente y para dar la noción de movimiento debemos borrar el cuadro anterior, de esta forma parecerá que el cuadro se desplaza y no que se dibuja

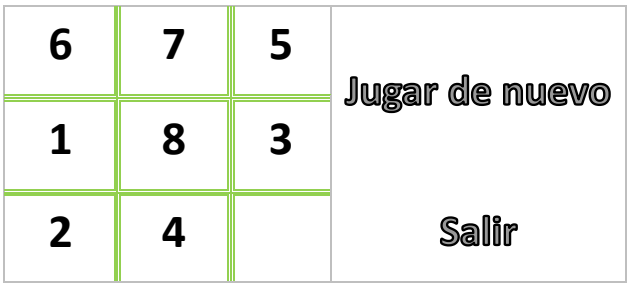
otro.

La función *movimiento,* evalúa si *get-key-press* es igual a alguna de las teclas de dirección, de serlo llama a *cuadro,* en la posición *x1*, *y1*, esta posición aumenta o disminuye en cada llamado dependiendo de la tecla de dirección presionada.

Nota: Los argumentos *lad* y *lad1*, corresponden al tamaño del lado del cuadro, es más útil usarlo como un argumento y no declararlo directamente en cada uso de la función *draw-solid-rectangle* porque de esta forma podremos variar el tamaño del cuadrado en cualquier momento con solo modificar el número correspondiente en el llamado de la función.

12.5 Actividades

1. Crear una ventana, esperar 3 segundos y cerrarla.
2. Crear una ventana y pintarla de azul.
3. Crear una ventana y pintarla de verde y luego limpiar todo su contenido.
4. Dibujar 10 píxeles de forma recursiva formando una línea recta.
5. Dibujar 5 líneas de forma recursiva en distintas partes de la ventana (Usar la función `(random)`).
6. Dibujar un cuadrado que cambie de color 10 veces. Usar la función `(sleep 0.5)` para observar el cambio de color y la función `random` para que cambie de color.
7. Dibujar 500 círculos pequeños de diferentes colores de forma recursiva en distintas partes de una ventana.
8. Dibujar un paisaje en una ventana.
9. Mostrar una imagen en una ventana, hacer que esta se mueva en la ventana con las teclas de dirección del teclado. Nota: Recuerden el ejemplo realizado en la página 81.
10. Hacer una función que espere a que se dé clic. Si el clic es el izquierdo imprimir “Se dio clic izquierdo” y debajo imprima las coordenadas donde se dio clic. Hacer lo mismo con el clic derecho y con el del centro, cambiando los mensajes.
11. Dibujar una matriz de 3x3 e importar dos imágenes a la misma ventana, una de las imágenes debe ser un texto que diga: “Jugar de nuevo” y la otra “Salir”.
12. Crear imágenes de los números del 1 al 8 y ubicarlos en cada uno de los cuadros de la matriz que creamos en el punto anterior, las imágenes de los números deben ubicarse aleatoriamente en los cuadros de la matriz, nunca en orden. Ejemplo:



13. Usando lo que se hizo en el punto anterior, luego llamar a una función que espere a que se dé un clic. Si se da un clic sobre los números que muestre un mensaje que diga "Números", si se da clic sobre el botón de "jugar de nuevo" que imprima un mensaje que diga "Ok" y si se da clic sobre "Salir" que se salga de la ventana. Con cada mensaje nuevo debe borrarse el viejo por lo que los mensajes se deben mostrar en la misma posición en una parte de la ventana. El programa no se debe cerrar hasta que se dé clic en "Salir". Basarse en los ejercicios de mouse enviados.
14. Con lo hecho de los puntos 11 y 12 crear una función que al presionar clic sobre uno de los números espere para recibir otro clic, si ese clic se dio en un lugar en blanco mover el número a ese lugar. La idea final es que el usuario pueda ordenar los números.
15. Desarrollar de una forma más eficiente el ejercicio número dos de la página 81.