Programación de datos con Racket

Editorial CIMTED

Autor:
Luis Eduardo Muñoz G.

ISBN: 978-958-53396-1-3

Editado en Colombia

Formato EPUB

2021

Programación de datos con RACKET

Luis Eduardo Muñoz Guerrero

Universidad Tecnológica de Pereira
Facultad de Ingenierías
Ingeniería de sistemas y computación

2021

Luis Eduardo Muñoz Guerrero
Profesor Titular
Universidad Tecnológica de Pereira

1ª Edición,
Editado en Colombia
Medellín - Abril 2021
ISBN: 978-958-53396-0-6 versión EPUB
ISBN: 978-958-53396-1-3 Versión PDF
Editado por el Centro Internacional de Marketing Territorial para la Educación y el desarrollo CIMTED

Cuidado de la Edición: Juliana Escobar Gómez
Calle 41 no 80B 120 Código postal 050010 Medellín, Antioquia - Colombia www.cimted.org
www.memoriascimted.com www.editorialcimted.com

Más informes editorialcimted@gmail.com,
Publicación electrónica editada en Colombia.

Página legal

Título: Programación de datos con Racket
ISBN Obra independiente: 978-958-53396-1-3
Sello editorial: Corporación Centro Internacional de Marketing Territorial para la Educación y el Desarrollo (978-958-53396)
Tema: Técnicas de programación
Materia: Programación. programas. datos de computadores
Tipo de Contenido: Computación y sistemas
Clasificación THEMA: Técnicas de programación
Público objetivo: Enseñanza universitaria o superior
Idioma: Español
Tipo de soporte: Libro digital descargable
Formato: Pdf (.pdf)
Tipos de acceso: Digital: descarga y online
Tamaño: 15Mb

"Editor: Corporación Centro Internacional de Marketing Territorial para la Educación y el Desarrollo. Corporación CIMTED Nit:811043395-0 editorialcimted@gmail.com "

Agradecimientos

A mis hijos Camilo, Luis y Alejandro; por siempre recordarme el significado de la palabra amor.

A mi hermana Sonia Muñoz, quien ha sido un pilar invaluable a lo largo de mi vida.

Agradezco de manera especial al estudiante Miguel Ángel López Fernández, por su apoyo en la recopilación de la información básica para el desarrollo de este libro.

Introducción

El siguiente documento presenta las implementaciones del paradigma de programación funcional con distintas estructuras de datos, al igual que varios servicios ofrecidos por el ambiente de programación Dr Racket.

Este libro está orientado a las personas que ya tienen algo de experiencia con los conceptos básicos de la programación como tipos de datos o estructuras de control; se elaboró de forma tal que proporcione la teoría necesaria para comprender los conceptos técnicos de cada aplicación, junto a un par de actividades complementarias que ayudarán a reforzar los conocimientos obtenidos. Igualmente, se compone de partes orientadoras, donde linealmente se desenvuelve la teoría con ejemplos generales.

Estructura de datos

Se explicará el funcionamiento de estructuras como listas, pilas y colas, árboles binarios, ficheros, entre otros.

1. Listas

1.1 Teoría sobre el funcionamiento de las listas

Las listas se definen como un conjunto de elementos de uno o más tipos de datos ordenados secuencialmente. Es una de las estructuras más flexibles en la programación ya que siempre permite modificar su tamaño, haciendo uso de operaciones básicas de inserción y eliminación de elementos.

Una lista se representa de la siguiente manera:

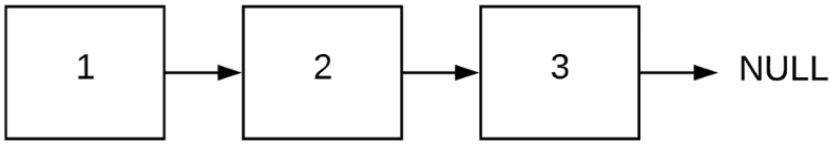


Ilustración 1. Representación de una lista

Se puede observar que cada elemento se sitúa uno tras otro, siguiendo un orden establecido. Hay cuatro casillas, cada una con un dato, las cuales se denominan **elementos** o nodos. El tamaño de una lista se compone a partir del número total de sus elementos, dicho tamaño puede cambiar sin importar la cantidad de memoria designada.

Los elementos pueden almacenar cualquier tipo de información y de la misma manera las listas pueden estar formadas por uno o varios tipos de datos. De lo anterior tenemos que los elementos se articulan de forma ordenada y escalonada obteniendo las listas enlazadas.

Cada elemento se conecta con el siguiente, al llegar al último este se dirige hacia la nada o vacío, al que también se le denomina *null*. Es útil para identificar la existencia de más elementos y por ende el tamaño de la lista.

1.2 Pares

Se le conoce como par a dos elementos ligados entre si. Se compone de una estructura donde el primer elemento se llama cabeza y el segundo se llama cola. Estas formas son creadas con el procedimiento *cons*; admiten todo tipo de dato permitiendo representar pilas, colas, árboles, ejecución de procesos e hilos, entre otros.

Una construcción con base en lo anterior sería lo siguiente:

```
1       (cons 4 2)
        (4 . 2)
```

Como resultado se mostrará un par de datos separados por un espacio, un punto y un espacio, de esa manera para todos los casos.

Se tienen las funciones prediseñadas, *car* y *cdr* las cuales servirán para determinar el elemento cabeza (aquel elemento situado de primero) y el elemento cola (aquel elemento situado de ultimo) respectivamente, así:

```
1       (car (cons 4 2))
2       (cdr (cons 4 2))
        4
        2
```

A partir de la creación de pares se pueden construir listas, como en el siguiente ejemplo:

```
1  (cons 5(cons 4 (cons 3 (cons 2 (cons 1 (cons 0 ()))))))
     (5 4 3 2 1 0)
```

1.3 Operaciones fundamentales sobre las listas

En el siguiente recuadro se muestran algunas de las funciones predefinidas en Dr Racket para el manejo de listas:

Función	Descripción
`(pair? V)`	Es una función con retorno de valores de tipo booleano. Se ingresan datos de cualquier tipo y se comprueba que sea una pareja.
`(list? V)`	Es una función con retorno de valores de tipo booleano. Se ingresan datos de cualquier tipo y se comprueba que sea una lista.
`(list? V)`	Es una función con retorno de valores de tipo booleano. Se ingresan datos de cualquier y se comprueba si ese dato está o no vacío (sin información)
`(length lista)`	Determina el tamaño de una lista.
`(list-ref lista pos)`	Retorna el elemento (ya sea lista o pareja) de una posición dada. Su posición se da en números enteros no negativos.
`(remove v lista proc)`	Eliminar un elemento de la lista en la posición (v – 1). Puede usar el proc (tipo bool) *como* comparador, aunque tiende a omitirse.

Función	Descripción
(reverse lista)	Invierte el orden de una lista.
(filter proc lista)	Según el procedimiento (positive?, even?,etc.) devuelve una lista con los elementos que lo cumplan.
(sort lista proc)	Ordena los elementos de una lista según el orden dado por proc (> o <).
(member pos lista)	Retorna la lista con el elemento buscado como primera posición. SI dicho elemento no existe se retorna #f.
(car lista)	Devuelve el primer elemento de una lista.
(cdr lista)	Devuelve una lista sin el primer elemento.
(list-set lista pos v)	Cambia el elemento de una lista en una posición v.
(split-at-right lista pos)	Divide una lista en dos, desde una posición dada.
(remove v lista proc)	Eliminar un elemento de la lista en la posición (v – 1). Puede usar el proc (tipo bool) *como* comparador, aunque tiende a omitirse.
(reverse lista)	Invierte el orden de una lista.
(filter proc lista)	Según el procedimiento (positive?, even?,etc.) devuelve una lista con los elementos que lo cumplan.

Tabla 1. Funciones para las listas Dr Racket

1.4 Ejercicios resueltos

Ejercicio No 1

Determinar el tamaño de una lista sin usar las funciones prediseñadas de Dr Racket.

```
1  (define mi-lista (list "cero" 1 "dos" 3 "cuatro" 5.322))
2
3  (define (tamano lista con)
4    (if(null? lista)
5       con
6       (tamano (cdr lista) (+ con 1))
7       )
8    )
9
10 (define (main)
11   (tamano mi-lista 0)
12   )
13 (main)
```

Análisis

En la línea número 1 se define *mi-lista* con 6 elementos, desde números enteros a datos de tipo cadena. En la línea 3 se crea la función *tamano*; el primer argumento recibirá todas las listas diseñadas por el usuario y el segundo hará la función de contador, es decir, contar el número de recursiones en realizadas, que a la vez será también el número de elementos.

Dentro de la función se encuentran dos condicionales, el primero verifica si la lista está vacía; si es verdadera se retorna el valor del contador y termina la recursión; si es falsa se llama a la función pero esta vez la lista irá sin el primer elemento e incrementará de uno en uno su valor.

Por último, en la línea 10 del código se declara una función main que será la encargada de ejecutar el programa.

El resultado es:

```
> 6
```

Ejercicio No 2

Dada una lista, solicitar un número por teclado y eliminarlo de dicha lista si este se encuentra presente.

```
1   (define (eliminar lista x)
2     (if(null? lista)
3        (lista)
4        (if(= (car lista) x)
5           (eliminar (cdr lista) x)
6           (cons (car lista)(eliminar (cdr lista )x))
7           )
8        )
9     )
10  (eliminar (list 1 2 3 4 5) 2)
```

Análisis

Se crea la función *eliminar* que recibirá una lista creada por el usuario y un entero *x* que será el número para eliminar. Dentro de la función se evalúa si la lista está vacía; si es verdad se crean parejas que se almacenarán en memoria mientras haya recursión; si es falso se evalúa la siguiente condición que verifica si hay el primer elemento de la lista es igual al dato ingresado. Si existe, se hace nuevamente el llamado a la función y se devuelve la lista con el número eliminado, caso contrario se crean pares

con cada elemento de la lista debido al llamado recursivo de la función.

El resultado de este programa es:

```
(1 3 4 5)
```

Ejercicio No 3.

Hacer un programa que reciba dos listas e intercale sus elementos.

```
1   (define (intercalar a b)
2     (if(and(null? a)(null? b))
3        (list)
4        (if(null? a)
5           (cons (car b) (intercalar a (cdr b)))
6           (cons (car a) (intercalar b (cdr b)))
7           )
8        )
9     )
10
11  (intercalar (list 1 3 5 7 11 13) (list 2 4 6 8))
```

Análisis

Este programa se construye a partir de intercalar elementos por ende, se deben obtener los 2 primeros datos de cada lista, posteriormente los segundos, y así sucesivamente hasta obtener cada elemento de las dos listas.

En la línea 1 se crea la función *intercalar* que recibe como argumentos dos listas, *a y b*. Dentro de la función se determina si *a* y *b* están vacías; si es verdad, se crea la lista final y el programa termina. Sin embargo, si es falso se evalúa nuevamente si ***a*** es nula; verdadero y se crean pares

del primer elemento de *b* debido a la llamada recursiva de la función con argumento (*cdr lista b*), realizando el cambio de lista con la cual interactuar. Por otro lado, si no se cumple, se realiza el mismo procedimiento que el anterior, pero para valores de ***a***.

El resultado de este programa es:

```
> (1 2 3 4 5 6 7 8 11 13)
```

1.5 Ejercicios de repaso

A continuación, se proponen algunos ejercicios que servirán como repaso de todo lo tratado en este capítulo.

1. Hacer una función recursiva que reciba una lista y retornar la suma de sus elementos.
2. Dada una lista compuesta por números enteros, retornar la sumatoria de todos los números pares.
3. Hacer una función recursiva que reciba una lista de enteros positivos y devolver el mayor valor contenido en la lista (o-1) si está vacía.
4. Hacer un programa que reciba una lista compuesta de números enteros e indicar si está ordenada ascendentemente.
5. Dada una lista desordenada de números enteros, ordenar dicha lista de mayor a menor (no utilizar la función sort). Se deben implementar los algoritmos de ordenamientos.
6. Dada una lista compuesta por cadenas, construir una nueva lista formada por los elementos que no estén repetidos. Ejemplo: (list “hola” “mundo” “mundo”) la nueva lista será (list “hola”).

7.Hacer una función que reciba una lista de números y un dato numérico, si este número no se encuentra en la lista, ingresarlo a ella en la parte final.

8.Hacer un programa que reciba nombres (cadenas de caracteres) y devuelva el que tenga mayor longitud.

9.Dada una lista compuesta por tres puntos ((list 1 2) es un punto donde x = 1-y = 2), retornar verdadero si dichos puntos forman un triángulo equilátero, y retornar falso en caso contrario.

10.Dada una lista compuesta por cadenas, retornar una nueva lista compuesta por las cadenas, pero sin sus vocales. Ejemplo: (list “hola” “mundo”) retornará (list “hl” “mnd”).

2. Pilas y colas

2.1 Pilas

(*Stack* en inglés) es una estructura de datos que permite almacenar y recuperar información a través del sistema de acceso LIFO (Last In First Out), donde el último elemento en entrar es el primero en salir.

El manejo de datos se realiza solo por un extremo en donde se ubica el último dato almacenado. Este será llamado TOS (top of stack o tope de pila). Mediante operaciones de inserción (push) y eliminación (pop) se tendrá acceso a cada información apilada en la estructura. Como ejemplo, si se tiene una pila de 4 elementos y se requiere el acceso al tercer elemento, solo basta con retirar el elemento ubicado en el tope.

Lo anterior se puede representar con la siguiente imagen:

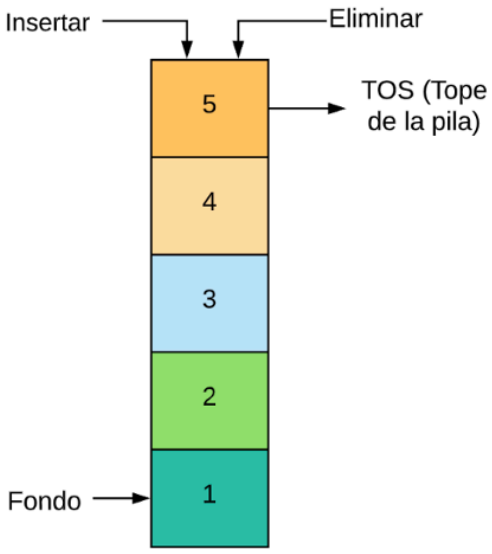


Ilustración 2. Representación de una pila

Las pilas suelen manejarse en los siguientes escenarios:

- Evaluación de expresiones en notación postfija
- Reconocedores sintácticos de lenguajes independientes del entorno
- Recursividad

2.2 Ejemplos con pilas

Se tiene el siguiente código:

```
1   (define (pila tam)
2
3   (define la-pila (make-vector tam))
4
```

```
5     (define i 0)
6
7     (define (pila-vacia?)
8         (if(= i 0)
9            #t
10           #f))
11
12      (define (pila-llena?)
13        (if(= i tam)
14           #t
15           #f))
16
17      (define (inserta dato)
18        (if(pila-llena?)
19           (error "Pila llena")
20           (begin
21             (vector-set! la-pila dato)
22             (set! i (+ i 1)))
23           )
24        )
25
26      (define (extrae)
27        (if(pila-vacia?)
28           (error "Pila vacia")
29           (begin
30             (set! i (- i 1))
31             (vector-ref la-pila i))
32           )
33        )
34
35    (define tabla (make-hash-table))
36    (hash-table-put! tabla 'pila-vacia? pila-vacia?)
37    (hash-table-put! tabla 'pila-llena  pila-llena?)
38    (hash-table-put! tabla 'inserta     inserta)
39    (hash-table-put! tabla 'extrae      extrae)
40    tabla)
41
42    (define (pila-vacia? pila)
43      ((hash-table-get pila 'pila-vacia?)))
44
45    (define (pila-llena? pila)
46      ((hash-table-get pila 'pila-llena?)))
47
48    (define (inserta pila dato)
49      ((hash-table-get pila 'inserta)dato))
50
51    (define (extrae pila)
52      ((hash-table-get pila 'extrae)))
53
54    (define pila1 (pila 10))
55
```

Pila es la función en la cual se llevarán a cabo cuatro operaciones que simularán el comportamiento de una pila en el sistema. Se crea un vector de tamaño *tam* y un índice *i* que guiará la posición y acceso de la información. Las operaciones son:

pila-vacia?: Encargada de verificar si el vector tiene información. Se basa en la posición del índice, donde, si

este es distinto de cero significa que se ha ingresado algún dato.

pila-llena?: Verifica si la pila está llena a través de la posición del índice, cuando este se iguala al tamaño del vector.

Inserta: Recibe como argumento un *dato* el cual será la nueva información. Verifica si la pila está llena, donde se ejecuta una línea de error si es verdadero, caso contrario se añade el *dato* a la posición actual de *i* y luego se incrementa su valor en uno.

Extrae: Muestra en pantalla la información de la última posición que tomo *i*.

Comenzando la línea 35 se crea la tabla hash y sus agrupaciones. Esto servirá para usar cada una de las funciones anteriores de manera independiente. Al terminar el programa principal se definen las funciones que conectaran la tabla hash con el ingreso de datos. Se hace el llamado final al programa principal y se interactúa con las funciones locales previamente definidas.

Como resultado se obtiene:

```
(pila-vacia? pila1)
> #t
(pila-llena? pila1)
> #f
(inserta pila1 45)
(inserta pila1 54)
(extrae pila1)
> 54
```

2.3 Colas

Es un grupo de elementos del mismo tipo los cuales son insertados en memoria a través de un extremo (llamado final) y eliminados a través de otro extremo llamado frente. En informática y redes se conocen como una estructura de datos con sistema FIFO (First In First Out), es decir, el primero en entrar es el primero en salir.

Esta dinámica se puede entender mejor con casos de la vida cotidiana como las filas en los supermercados o la impresión de documentos; los dos centran sus procesos en el momento en que va llegando la información, la persona o documento que se posicione de primero, será de la misma manera lo primero en salir.

La representación en memoria del concepto anterior es:

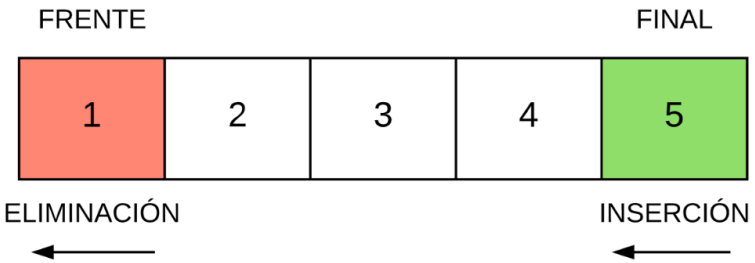


Ilustración 3. Representación de una cola

Este modelo de estructura puede usarse en sistemas informáticos para el almacenamiento y transporte de información, programación orientada a objetos, entre otros. De igual manera se destacan sus propósitos en la programación funcional, como una recursión más eficiente.

2.4 Recursión por Cola

Se tiene el siguiente código:

```
1    (define (tamaño lista)
2      (if(null? lista)
3      0
4      (+ 1 (tamaño (cdr lista)))
5      ))
```

Al llamar la función se da el siguiente proceso

```
6              (tamaño (list 1 2 3 4 5))
```

```
1    (tamaño (1 2 3 4 5))
2    (+ 1 (tamaño (2 3 4 5))
3    (+ 1 (+ 1 (tamaño (3 4 5))))
4    (+ 1 (+ 1 (+ 1 (tamaño (4 5)))))
5    (+ 1 (+ 1 (+ 1 (+ 1 (tamaño (5))))))
6    (+ 1 (+ 1 (+ 1 (+ 1 (+ 1 (0))))))
7    (+ 1 (+ 1 (+ 1 (2)))))
8    (+ 1 (+ 1 (3))))
9    (+ 1 (4))
10   (5)
```

Se puede observar que primero se acumulan las cantidades a sumar, y las sumas se posponen hasta tener la lista vacía. Esto es un procedimiento recursivo natural el cual compromete la acción del procesador en la memoria de un computador, por lo que en casos de mucha información se puede correr el riesgo de tener un desbordamiento de memoria o "*stack overflow*".

Viendo lo anterior de una manera distinta:

```
(define (tamaño lista)
  (define (aux l cont)
    (if(null? l)
       cont
       (aux (cdr l) (+ 1 cont)))
    )
  (aux lista 0)
  )
```

Se crea una función local *aux* la cual será la encargada de iterar sobre la lista para contar el número de elementos; *cont* ira sumando uno cada nuevo llamado de la función.

Su funcionamiento es el siguiente:

```
(tamaño (list 1 2 3 4 5))
(aux (1 2 3 4 5) 0)
(aux (2 3 4 5) 1)
(aux (3 4 5) 2)
(aux (4 5) 3)
(aux (5) 4)
(aux ( ) 5)
(5)
```

Se obtiene el mismo resultado, pero con la diferencia de que la recursión se hace dentro de una función, por lo que los llamados y asignaciones de espacio para cada dato o proceso se harán dentro de un espacio de memoria definido previamente por la función principal. Nótese que la suma de la cantidad de elementos se realiza a la par de cada nuevo llamado de *aux*, evitando postergar los procesos.

De lo anterior, se puede concluir que en el lenguaje Dr Racket para el uso de la recursión basada en colas la cantidad de memoria utilizada es fija para cada ejecución.

2.5 Ejercicios de repaso

Se proponen a continuación una serie de ejercicios para afianzar los conceptos vistos en este capítulo.

1.Calcular la serie de Fibonacci implementando los modelos de pilas y colas

2.Usando las colas, calcular la sumatoria de n números.

3.Realizar la multiplicación de números sin usar las funciones prediseñadas. Utilice las colas para la recursión.

4.Hacer un programa que calcule los números primos de una lista y guarde los datos en una pila. También debe introducir un número a buscar en la pila, para luego mostrar en pantalla la información almacenada hasta llegar al dato pedido.

5.Ordenar un vector de números enteros teniendo en cuenta el funcionamiento de las pilas.

6.Usando las pruebas de escritorio, describir el funcionamiento de selección de un elemento usando las pilas y las colas.

Nota: se puede utilizar la libreria (require data/queue) para utilizar funciones representativas del sistema FIFO.

3. Tablas hash

3.1 Concepto

También llamada matriz asociativa, tabla de dispersión o tabla fragmentada, es una estructura que sirve para el almacenamiento de datos. Relaciona valores llamados llaves (conocidos también como claves) con otros denominados valores. Maneja operaciones de inserción y búsqueda, siendo esta última la que permite el acceso a cada uno de los datos en tiempo de ejecución.

Una tabla hash se representa de la siguiente manera:

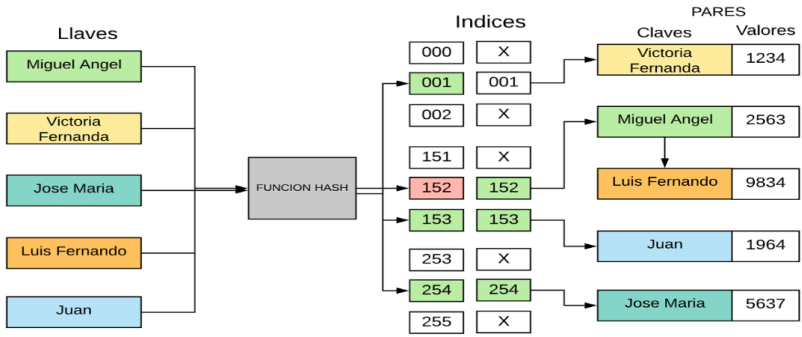


Ilustración 4. Representación de una tabla hash

Lo anterior representa una estructura con un conjunto de datos (nombres y la asignación de valores numéricos). Este se almacena en un vector de n dimensiones donde internamente se toma y posiciona la información a través de una función denominada *función hash*, ésta aleatoriamente indexa cada clave con su valor creando al final un modelo de tabla organizado.

3.2 Función hash

También conocidas como funciones resumen o funciones dispersoras, son funciones que utilizan un algoritmo matemático que calcula el índice adecuado para cada elemento (o llave) para transforman un conjunto de datos en un código alfanumérico con una longitud fija (este último es independiente del tamaño del conjunto de datos).

Dado que calculan la posición para cada elemento, al momento de requerir una información específica basta con solo identificar la llave necesaria.

Gracias a esto las tablas hash pueden destacar la eficiencia como factor sobresaliente de otras estructuras como los arreglos, listas, entre otros.

Algunas de las funciones básicas para la creación y función de estas tablas son:

Función	**Ejemplo**	**Descripción**
(hash clave valor)	`(hash "juan" 1234)`	Crea una tabla hash inmutable, con igual número de claves y valores
(hash-equal? v)	`(hash-equal? tabla)`	Evalúa si es una estructura hash
(hash? v)	`(hash? tabla)`	Verifica si un elemento es una tabla hash

(make-hash v) (make-hash-table)	`(define tablas (make-hash))` `(define tablas (make-hash-table))`	Dos maneras distintas de crear una tabla hash mutable. La diferencia se encuentra en la intensidad de comparación entre claves y valores. Cuando los dos son completamente iguales (y se dirigen a un mismo dato) se recomienda usar la segunda función.
hash-set! hash-table-put!	`(hash-set! tablas "a" (list 1 2 3))` `(hash-table-put! tablas "a" (list 1 2 3))`	Dos formas para agregar nuevos grupos a una tabla hash. La segunda función formaría parte de la estructura dada por make-hash-table.
hash-ref hash-table-get	`(hash-ref tablas "a")` `(hash-table-get tablas "a")`	Devuelve el valor que le corresponde a una clave indicada, si existe. De igual forma, el segundo hace parte de la estructura make-hash-table
hash-remove	`(hash-remove! tablas "a")`	Elimina una clave y su respectivo valor en tabla
hash-count	`(hash-count tablas)`	Devuelve el tamaño de la tabla hash creada, a partir del número de pares contenidos

Tabla 2. Funciones de manejo para tablas hash

3.3 Ejemplos

```
1   (define tabla (make-hash))
2
3   (hash-set! tabla "Juan" 1234)
4   (hash-set! tabla "Pedro" 4321)
5   (hash-set! tabla "Compras" (list "Manzanas" "Peras"
    "Tomates"))
6   tabla
7   (hash-count tabla)
8   (hash-ref tabla "Pedro")
9   (hash-remove! tabla "Juan")
10  tabla
```

Análisis

En la primera línea se crea una tabla hash con los respectivos comandos. Posteriormente se ingresan datos; nótese que en la línea 5 el valor asignado a la llave *compras* es una lista de tres elementos, donde cada uno es un dato de tipo cadena. En la línea 6 se hace el llamado a *tabla* para que esta se imprima en la línea de comandos. Las líneas 7,8 y 9 utilizan funciones de conteo, referencia y eliminación de elementos respectivamente. Al final se muestra nuevamente la tabla con los cambios realizados.

Se tiene como resultado:

```
>  (make-hash
    (list
     (cons "Juan" 1234)
     (cons "Pedro" 4321)
     (cons "Compras" (list "Manzanas" "Peras" "Tomates"))))
>  3
>  4321
>  (make-hash (list (cons "Pedro" 4321) (cons "Compras" (list
   "Manzanas" "Peras"
```

4. Árboles binarios

4.1 Conceptos de árboles

Un árbol binario es una estructura ramificada compuesta de varios elementos. Se pueden forman a partir de pares y listas, aunque existen modelos con vectores o estructuras de registros.

En recursión se definen como la estructura formada por un elemento y dos árboles, como en la siguiente ilustración:

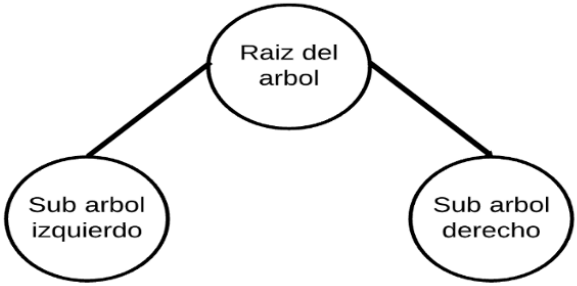


Ilustración 5. Estructura de un árbol binario

Los arboles pueden clasificarse en dos grupos, los que almacenan información conocidos como nodos internos, y los que se encuentran vacíos, o nodos externos (llamados también hojas). Todo elemento perteneciente al árbol se conocerá cómo nodo. Existirá un elemento raíz (nodo raíz) que despliega el resto de los nodos en la estructura, los cuales se conocerán cómo hijos. Cada nodo podrá almacenar de cero a dos hijos como máximo los cuales estarán guiados en el lado izquierdo y derecho de cada dato.

Este modelo de estructuras beneficia el proceso de búsqueda e identificación de elementos específicos. De igual forma se hace útil para la resolución de problemas como representación de fórmulas matemáticas, circuitos, organigramas, bases de datos, estructuras de condición, entre otras.

Tipos de recorrido

Existen tres modelos de recorrido para arboles binarios :

Nota: Este proceso se sigue recursivamente en cada paso.

Pre-orden: Examinar la raíz (nodo actual) ,recorrer la rama izquierda en pre-orden, recorrer la rama derecha en pre-orden.

In-orden: Recorrer la rama izquierda en in-orden, examinar la raíz (nodo actual),recorrer la rama derecha en in-orden.

Post-orden: Recorrer la rama izquierda en post-orden, recorrer la rama derecha en post-orden, examinar la raíz (nodo actual).

4.2 Ejercicios resueltos

A continuación, se mostrarán algunos programas para la creación de árboles binarios en Dr Racket. El modelo implementado es con estructuras de registros, sin embargo, se invita al alumno a determinar nuevas formas de generar dichas estructuras.

Ejercicio No 1

Crear un árbol binario en Dr Racket y ejemplificar sus recorridos manualmente

```
1  (define-struct arbol (raiz izq der))
2
3  (define t(make-arbol 'A
4            (make-arbol 'C empty empty)
5            (make-arbol 'B
6                        (make-arbol 'D empty empty)
7                        (make-arbol 'E empty empty))))
```

A partir de las funciones de tipo struct se crea una estructura de tipo árbol con campos *raiz*, *izq* y *der* las cuales darán guía de los nodos a posicionar. Posterior al modelo definimos una función *t* donde se configuran cada uno de los hijos, empezando con el dato *'A* como raíz. De manera gráfica, el código ejemplifica lo siguiente:

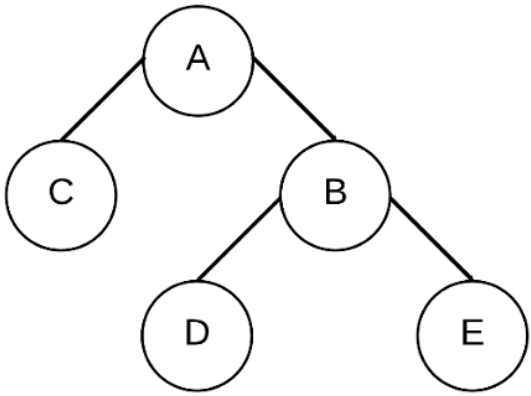


Ilustración 6. Representación de un árbol binario

Para identificar los elementos que lo componen se usaron las siguientes funciones:

```
1     (define (hder a)
2       (arbol-der a))
3     (hder t)
4
5     (define (hizq a)
6       (arbol-izq a))
7     (hizq t)
8
9     (define (raiz a)
10      (arbol-raiz a))
11    (raiz t)
12
13    (define (hoja? a)
14      (cond
15        [(empty? a) #f]
16        [(and(empty? (hder a))
17             (empty? (hizq a))) #t]
18        [else #f]))
19    (hoja? (hizq t))
```

Note que para saber si un elemento del árbol es o no una hoja se debe antes verificar si está o no vacío (un nodo hoja no puede alojar información en ninguno de sus extremos). Nuevamente sí es falso, se concluye que la estructura no está vacía.

Como resultado de este programa se tiene:

```
>  (make-arbol 'B (make-arbol 'D '() '()) (make-arbol 'E '()
   '()))
>  (make-arbol 'C '() '())
>  'A
>  #true
```

El recorrido del árbol en sus tres clasificaciones es:

Pre-orden: A C B D E.
In-orden: C A D B E.
Post-orden: C D E B A.

Ejercicio No 2

Hacer un programa que reciba un árbol y retorne el número de elementos que lo componen.

```
1  (define (contarEle a)
2    (cond
3      [(empty? a) 0]
4      [else (+ 1 (contarEle (hizq a))(contarEle (hder a)))])
5    )
6  (contarEle t)
```

A través de recursión se hace el conteo de elementos con una función que recibe como argumento un árbol evaluando si está o no vacía. Siendo falso se hace un nuevo llamado a la función, pero esta vez con los arboles de izquierda y derecha compuestos a partir de A (usando las funciones previamente definidas).

Como resultado de este programa se tiene:

```
>  5
```

Ejercicio No. 3

Determinar el mayor y menor número de un árbol binario

```
1   (define a (make-arbol 3
2                         (make-arbol 4 empty empty)
3                         (make-arbol 0
4                                     (make-arbol 1 empty empty)
5                                     (make-arbol 2 empty empty))))
```

```
6   (define (mayor a)
7     (cond
8       [(hoja? a)(raiz a)]
9       [(empty? (hizq a)) (max (raiz a) (mayor (hder a)))]
10      [(empty? (hder a)) (max (raiz a) (mayor (hizq a)))]
11      [else (max(raiz a)(mayor (hizq a))(mayor (hder a)))])
12    )
13
14  (define (menor a)
15    (cond
16      [(hoja? a)(raiz a)]
17      [(empty? (hizq a)) (min (raiz a) (menor (hder a)))]
18      [(empty? (hder a)) (min (raiz a) (menor (hizq a)))]
19      [else (min (raiz a)(menor (hizq a))(menor (hder a)))])
20    )
```

Se escribe una nueva estructura de árbol compuesta por números enteros. Las funciones de *menor* y *mayor* reciben un árbol como argumento. Dentro de cada función se evalúa si la estructura está o no vacía; siendo verdadero se retorna la información del nodo raíz, siendo falso se evalúa si el nodo izquierdo o el nodo derecho están vacíos; si se cumple la condición se retorna el mínimo entre la raíz y la llamada recursiva de la función (sus nodos izquierdo y derecho) que tiene como argumento el árbol, pero con un elemento (o nivel) fuera.

Si ninguna de las sentencias anteriores se cumple se determina el mínimo entre el nodo raíz de cada llamada a función y la recursión para los lados izquierdos y derechos, lo que permite avanzar primero por el lado izquierdo y luego por el derecho.

Como resultado se obtiene:

```
>  4
>  0
```

4.3 Arboles ligeros y perezosos

Para facilitar el uso y creación de árboles binarios, Dr Racket provee la librería *datos/lazytree*. En esta se proponen funciones de operaciones básicas como la creación rápida de árboles, filtración de información, recorridos, entre otras.

Algunas de dichas funciones son:

Nota: Luego de instalar el paquete, iniciarlo con (require data/lazytree)

Función	Ejemplo	Descripción
make-tree	(make-tree <secuencia de datos> <nodo raiz> <#:with-data> <#:empty-pred>)	Crea un árbol a partir de un nodo raíz. Recursivamente despliega la secuencia de datos.
tree-traverse	(tree-traverse <árbol o secuencia de datos> <#:order> <#:converse? >)	Recorre un árbol en orden de tipo 'pre, 'post, 'ino 'level.
tree-map	(tree-map <procedimiento> <árbol o secuencia de datos>)	Recorre un árbol por cada elemento, aplicando un procedimiento.
tree-filter	(tree-filter <sentencia booleana> <árbol o secuencia>)	Filtra todos los elementos que cumplan con la sentencia dada.

export-tree	(export-tree <procedimiento> <árbol o secuencia>)	Exporta un árbol binario.

Tabla 3. Funciones de la librería Lazytree.

Ejemplo No 1

Crear un árbol

```
1  #lang Dr Racket
2
3  (require data/lazytree)
4
5  (struct taxon (name children))
6  (define dog (taxon "Dog" '()))
7  (define cat (taxon "Cat" '()))
8  (define mammal (taxon "Mammal" (list dog cat)))
9
10 (export-tree list (make-tree taxon-children
11                                mammal
12                      #:with-data taxon-name))
```

Luego de implementar la librería, se crea una estructura *taxón* con campos *name* y *children*; se crean los modelos *dog, cat* y *mammal* a partir de la estructura inicial. Al final se exporta un árbol en forma de lista usando como secuencia el campo *children* de la estructura *taxon* y usando como nodo raíz el modelo *mammal*; *with-data* especifica la información de cada nodo, que en este caso están compuestos por los nombres puestos en la estructura.

Se tiene como resultado:

```
> '("Mammal" ("Dog") ("Cat"))
```

Ejemplo No 2

Usando el árbol anterior, usar las funciones de recorrido y mostrar el resultado:

```
1  (require data/lazytree)
2  (require relation/transform)
3  …
4  (define t (make-tree taxon-children
5                                    mammal
6                         #:with-data taxon-name))
7
8  (->list (tree-traverse #:order 'pre t))
9  (->list (tree-traverse #:order 'post t))
10 (->list (tree-traverse #:order 'in t))
11 (->list (tree-traverse #: order 'level t))
```

Para la representación de la información del árbol mediante esta librería se requiere el uso de *relation/ transform*.

Usando la estructura de árbol anterior, se crea una función *t* que crea el nuevo árbol; se proponen los recorridos de *preorden, postorden, inorden* y por *niveles*; la información generada esta en forma de flujo de información, por lo que para convertirla en un formato de listas representativo, debe de usarse *->list*.

Como resultados se obtienen:

```
> '("Mammal" "Dog" "Cat")
> '("Dog" "Cat" "Mammal")
> '("Dog" "Mammal" "Cat")
> '("Mammal" "Dog" "Cat")
```

Nota: Los casos anteriores fueron tomados de la documentación oficial de Dr Racket, por lo que se recomienda

profundizar en el manejo de las funciones anteriores a través de la información de cada librería.

4.4 Problemas de repaso

Se proponen a continuación una serie de ejercicios para afianzar la teoría explicada en este capítulo.

1. Crear una función que reciba un árbol binario y agregue a este un nuevo nodo.

2. Crear una función que reciba un árbol binario y un dato. Recorrer el árbol y eliminar el dato ingresado.

3. Crear una función que reciba un árbol binario con números enteros. Usar los tres tipos de recorrido (pre-orden, in-orden, post-orden) y mostrar en pantalla el resultado.

4. Crear una función que reciba un árbol y lo organice de menor a mayor.

5. Definir una función que reciba un árbol de números enteros. Esta deberá retornar una nueva estructura en la que cada nodo después de la raíz tendrá una lista de tres elementos, el primero y el segundo serán los hijos de ese nodo y el tercero será la suma de los anteriores. Si el nodo es una hoja, en lugar de una lista se debe ingresar un cero.

6. Crear una función que determine cuantas raíces no nulas tiene un árbol binario. Una raíz no nula es aquella que tiene por lo menos un hijo (ya sea de izquierda o derecha).

5. Estructuras

5.1 Teorías de las estructuras de datos

Son las distintas formas de organizar y representar los datos almacenados en memoria. Algunos modelos de estructuras son:

- **Arreglos** (vectores): Conjunto finito de datos compuestos de un solo tipo organizados consecutivamente. Son estáticos ya que se define el tamaño a utilizar.
- **Listas**: Conjunto dinámico de datos (almacena distintos tipos).Su tamaño puede variar antes o después del tiempo de ejecución.
- **Pilas** (Stack): Listas en las que la inserción y borrado de elementos se realiza solo por un extremo. Denominados también sistemas LIFO (Last In First Out).
- **Colas:** Listas en las que la inserción de elementos se realiza al final y la eliminación al comienzo. Llamadas también sistema FIFO (First In First Out)
- **Árboles binarios:** Estructura que representan la relación entre dos o más elementos de un conjunto.

En Dr Racket las estructuras se manejan a través de registros, que se son tipos de datos que almacenan a otros llamados campos; estos últimos pueden ser de tipo entero, flotante, cadena, etc. Este manejo permite acoplarse con las formas anteriores, permitiendo un uso más eficiente de la información. Gracias a ello se logra la creación de programas capaces de calcular gran volumen de información de manera organizada.

En Dr Racket las estructuras se manejan a través de registros, que se son tipos de datos que almacenan a otros llamados campos; estos últimos pueden ser de tipo entero, flotante, cadena, etc. Este manejo permite acoplarse con las formas anteriores, permitiendo un uso más eficiente de la información. Gracias a ello se logra la creación de programas capaces de calcular gran volumen de información de manera organizada.

Para el manejo de datos se tienen operaciones básicas como:

- **Recorrido:** permite acceder a cada registro (o elemento) procesado.
- **Búsqueda:** Encuentra un registro o elemento de acuerdo con un criterio (clave).
- **Inserción:** Añade nueva información, ya sea un registro o elemento.
- **Eliminación:** Operación de borrado de datos específicos
- **Ordenación:** Clasifica los elementos (o registros) de acuerdo con un orden lógico

5.2 Funciones para el manejo de estructuras

Para la creación de estructuras se usa:

Función	Ejemplo	Descripción
(define-struct <nombre-estructura>(<nombre-campo>))	(define-struct bool (a b))	Crea un modelo de estructura opaca (solo muestra el nombre de la estructura cuando se le llama).
make-<nombre-estructura>	(define cond (make-bool 0 1))	Es una función constructora, es decir, define nuevas funciones a partir de un modelo ya diseñado.
<nombre-estructura>?	(bool? cond)	Evalúa si la función creada hace parte del modelo de estructura.
<nombre-estructura>-<nombre-campo> <función-constructora>	(bool-b cond)	Retorna el valor de un campo, asignado por una función constructora.
(define-struct (<nombre-estructura><estructura-madre>) (<nombre campo>))	(define –struct(pqbool bool (c d))	Crea una subestructura a partir de una <estructura madre>
(define-struct <nombre-estructura> (<nombre-campo>) #:transparent)	(define-struct bool (a b) #:transparent)	Define una estructura de tipo transparente, en la que se muestran todos los campos que la componen junto a su información
(define-struct <nombre-estructura> (<nombre-campo>) #:mutable)	(define-struct bool (a b) #:mutable)	Crea una estructura donde los campos pueden ser alterados en cualquier instante.
set-<nombre-estructura>-<nombre-campo>!	(set-bool-a! cond 67)	Como la función constructora, crea una determina los valores para cada campo.

Tabla 4. Funciones para el manejo de estructuras

5.3 Ejercicios resueltos

Ejercicio No 1

Realice un programa que le permita ingresar día, mes y año. Guarde los valores en dos estructuras, una llamada nacimiento y otra llamada fecha actual

```
(define-struct fecha (dia mes ano))

(define (main)
  (display "Ingrese una fecha de nacimiento: \n")
  (define nacimiento (make-fecha
                       (begin(display "Dia: ")(read))
                       (begin(display "Mes: ")(read))
                       (begin(display "Año: ")(read)))
    )

  (display "Ingrese la fecha actual: \n")
  (define fecha_actual (make-fecha
                       (begin(display "Dia: ")(read))
                       (begin(display "Mes: ")(read))
                       (begin(display "Año: ")(read)))
    )
  (display "La fecha de nacimiento es: ")
  (begin
    (display (fecha-dia nacimiento))
    (display "/")
    (display (fecha-mes nacimiento))
    (display "/")
    (display (fecha-ano nacimiento)))
  (display "\n")
  (display "La fecha actual es: ")
  (begin
    (display (fecha-dia fecha_actual))
    (display "/")
    (display(fecha-mes fecha_actual))
    (display "/")
    (display (fecha-ano fecha_actual)))
  )
(main)
```

Análisis.

Primero se define el modelo de estructura *fecha* encargado de almacenar los datos de *nacimiento* y

fecha_actual. Luego se crea una función main donde se definen las funciones constructoras para los dos tipos de fechas requeridos (líneas 5 y 12). En las siguientes líneas se pide por teclado llenar la información para *día*, *mes* y *ano*. Después de almacenar la información se visualiza en pantalla cada función a través del llamado del modelo *fecha*.

Este ejercicio tiene como resultado:

```
Ingrese una fecha de nacimiento:
Dia: 12
Mes: 06
Año: 2003
Ingrese la fecha actual:
Dia: 29
Mes: 06
Año: 2020
La fecha de nacimiento es: 12/6/2003
La fecha actual es: 29/6/2020
```

Ejercicio No.2

Realizar un programa que pida en tiempo de ejecución los nombres y salarios de un cierto número de personas. Al final deberá mostrar cuales son las personas con un salario por encima del promedio y cuales por debajo.

```
1   (define-struct nomina (nombre sueldo))
2
3   (define (ver trabajador prom)
4     (if(> (nomina-sueldo trabajador) prom)
5        (display (nomina-nombre trabajador))
6        )
7     )
8   (define (ver2 trabajador prom)
9     (if(< (nomina-sueldo trabajador) prom)
10       (display (nomina-nombre trabajador))
11       )
12    )
13
14  (define (main)
15    (display "NOMINA \n")
16    (define n1 (make-nomina
17                 (begin(display "Nombre: ") (read))
18                 (begin(display "Sueldo")(read)))
19      )
20    (display "\n")
21    (define n2 (make-nomina
22                 (begin(display "Nombre: ") (read))
23                 (begin(display "Sueldo")(read)))
24      )
25    (display "\n")
26    (define n3 (make-nomina
27                 (begin(display "Nombre: ") (read))
28                 (begin(display "Sueldo")(read)))
29      )
30    (display "\n")
31      (define n4 (make-nomina
32                 (begin(display "Nombre: ") (read))
33                 (begin(display "Sueldo")(read)))
34      )
35    (display "\n")
36    (display "El promedio de sueldos es: ")
37    (define prom (ceiling (/ (+ (+ (nomina-sueldo n1) (nomina-sueldo n2))(+
    (nomina-sueldo n3)(nomina-sueldo n4))) 4)))
38    (display prom)
39    (display "\n")
40    (display "Trabajadores con salario mayor al promedio:\n")
41    (ver n1 prom)
42    (ver n2 prom)
43    (ver n3 prom)
44    (ver n4 prom)
45    (display "Trabajadores con salario menor al promedio:\n")
46    (ver2 n1 prom)
47    (ver2 n2 prom)
48    (ver2 n3 prom)
49    (ver2 n4 prom)
50    )
51  (main)
```

Análisis.

Se crea el modelo de estructura *nomina* con campos *nombre* y *sueldo*. En las siguientes líneas se construye las funciones *ver* y *ver2* que servirán como evaluadores al momento de determinar si una persona está por debajo o por encima del promedio de sueldos.

Dentro de una función main se crean las funciones constructoras para cada trabajador (en este caso se escogieron 4 personas). Posteriormente se declara la variable *prom* la cual calculará el promedio total de los sueldos usando el llamado a estructura (*nomina-sueldo trabajador*).Por último, se clasifican las personas que se encuentran por debajo y por encima de *prom*, haciendo un llamado a la función *ver* para determinar quienes se encuentran por encima y *ver2* de igual forma para quienes estén por debajo.

Un posible resultado para el ejercicio puede ser:

```
> NOMINA
  Nombre: Miguel
  Sueldo123

  Nombre: Jose
  Sueldo456

  Nombre: Lina
  Sueldo789

  Nombre: Ester
  Sueldo756

  El promedio de sueldos es: 531

  Trabajadores con salario menor al promedio:
  Lina
  Ester
  Trabajadores con salario menor al promedio:
  Miguel
  Jose
```

5.4 Ejercicios de repaso

Se proponen a continuación una serie de ejercicios para afianzar la teoría explicada en este capítulo.

1.Elaborar una función que reciba en tiempo de ejecución tres coordenadas del plano cartesiano, cada

punto será una estructura. Al final deberá retornar una lista con los 3 puntos organizados mayor a menor a partir del concepto de distancia al origen.

2.Elaborar una función que reciba como parámetro una estructura llamada persona, esta devolverá una estructura derivada llamada trabajador. Deberá pedir en tiempo de ejecución el NIT y el número de teléfono.

3.Realizar un programa que reciba como argumentos una lista de estructuras (personas), la cual contara también con un campo nombre; el programa debe retornar nuevamente una lista de estructuras ordenadas alfabéticamente (solo por iniciales).

4.Realizar un programa que reciba una lista de estructuras de tipo fecha, y que retorne una lista ordenada con las fechas anteriores a posteriores. Si algún elemento no existe o no es del tipo fecha, retornar una lista vacía.

5.Realizar un programa que arroje un menú con las siguientes opciones:
-Equipo ganador
-Historial de partidos

Con ello deberá realizar funciones que reciban como parámetros estructuras de tipo partido, con campos equipo y goles; para la primera opción se tendrá que mostrar el equipo ganador del total de partidos jugados; en la segunda deberá mostrar los partidos ordenados descendentemente.

6. Archivos.

6.1 Funciones básicas

Los archivos o ficheros basan su funcionamiento a partir de puertos o flujos de información. Más en detalle, se dividen en puertos de ***entrada*** donde se puede leer información de algún lugar específico y puertos de ***salida*** donde se puede escribir y almacenar información.

Para el manejo de este campo se utilizan 3 funciones principales:

`Open-input-file` - Abre un archivo para lectura

Su estructura es:

```
(open-input-file <ruta> [#:mode {'binary | 'text}])
```

Tiene como base el modo `'binary`.

`Open-output-file` - Abre un archivo para escritura

Su estructura es:

```
(open-output-file <ruta> [{#:mode {'binary | 'text}}]? [{#:exists {'error|'append|'update|'can-update|'replace|'truncate|'must-truncate}}]?)
```

El campo `#:exists` indica las acciones a seguir luego de verificar que el archivo existe. Cada una significa:

- `'error`: Arroja un error si el archivo ya existe.
- `'replace`: Reemplaza un archivo existente y crea uno nuevo.
- `'truncate`: Remueve todos los datos antiguos del archivo.
- `'must-truncate`: Elimina todos los datos antiguos de un archivo existente. Si el archivo no existe se lanza un error.
- `'append`: Posiciona el cursor al final del archivo ya sea que exista o no información.
- `'update`: Abre un archivo existente sin truncarlo, si no existe lanza un error.
- `'can-update`: Abre un archivo existente sin truncarlo, o también puede crear el archivo si este no existe.

`Open-input-output-file` - Abre un archivo para lectura y escritura

Su estructura es:

```
(open-input-output-file <ruta> [{#: mode
{'binary|'text}}]? [{#: exists
{'error|'append|'update|'replace|
'truncate}}]?)
```

Se tienen algunas funciones que amplían las actividades de los archivos, como las siguientes:

Función	Ejemplo	Descripción
port?	(port? v)	Evalúa si el dato es un puerto de cualquier tipo.
input-port?	(input-port? v)	Función de tipo booleana que determina si el argumento es un archivo de tipo lectura.
output-port?	(output-port? v)	Función de tipo booleana que determina si el argumento es un archivo de tipo escritura.
close-<tipo de archivo>-port	(close-input-port v) (close-output-port v)	Cierra un archivo creado previamente.
file-exists? <archivo>	(file-exists? v)	Función booleana que evalúa si v es un fichero (archivo).
delete-file	(delete-file v)	Elimina un fichero creado.
eof-object? <archivo>	(eof-object? v)	Verifica si se termina el flujo de información almacenado en el archivo.
read <archivo>	(read “archivo.txt”)	Devuelve un elemento del archivo
read-char <archivo>	(read-char “archivo.txt”)	Muestra el primer carácter almacenado en el archivo.
display <dato> <archivo>	(display “hola” “archivo.txt”)	Muestra el dato, pero esta vez en el fichero.
write-char <char> <archivo>	(write-char #\p “archivo.txt”	Escribe un carácter dentro del archivo
port?	(port? v)	Evalúa si el dato es un puerto de cualquier tipo.

Tabla 5. Funciones para el manejo de ficheros

Nota: Se le recomienda al alumno visitar la documentación oficial para ampliar las funciones referentes al manejo de archivos (puertos o ficheros).

6.2 Ejemplos resueltos

Ejercicio No 1

Realizar un programa que reciba un fichero como argument y cargue la información almacenada en la consola de Dr Racket.

```
1  (define p (open-input-file "formato.txt"))
2  (define (f x)
3    (if(eof-object? x)
4       (begin
5         (close-input-port p)
6         (list))
7       (cons x (f (read p)))))
8
9  (f p)
```

Análisis

Se crea el archivo en la misma ruta en la que se guarde el documento del código. También se puede proporcionar como argumento de *open-input-file* la ruta entera del archivo de texto si estos dos están en directorios distintos.

En la línea 1 se crea una función la cual almacenará el fichero de entrada *"formato.txt"*; la función *f* recibe como argumento un archivo. Al momento de la ejecución se evalúa si x devolverá un *eof-object?* (final de línea), si es verdadero significa que no hay más flujo de información, por lo que se cierra el archivo para luego mostrar una lista con todos los datos leídos. Si es falso se crean pares con los

elementos tomados por x y el nuevo llamado a la función la cual usa como argumento (*read p*), lo que genera que se vaya leyendo cada carácter existente en el archivo.

Como resultado se debe de obtener:

```
> (#<input                port:/Users/miguellopez/Desktop/LIBRO
  programación/formato.txt>
  Hola como estas se llama Angela)
```

Ejercicio No 2.

Realizar un programa que guarde una lista en un archivo de texto. Debe de haber un salto de línea por dato de la lista.

```
1    (define a (open-output-file "formato2.txt"))
2    (define (p a x)
3      (if(not(null? x))
4         (begin
5           (display (car x) a)
6           (newline a)
7           (p a (cdr x)))
8         (begin
9           (list)))
10     (close-output-port a))
11
12   (p a (list 1 2 3 4 5 6 7 8 9 10))
```

Análisis

Se crea la función *a* que contendrá el archivo de salida "*formato2.txt*".La función *p* tiene como argumentos *a* que recibe un archivo, y *x* que será la lista para guardar. Se evalúa si la lista ingresada es nula, esto, precedido de un factor booleano *not* que dará el resultado contrario de la validación. Si la condición arroja verdadero, el primer

elemento de la lista se ingresa en el archivo y luego de un salto de línea se realiza el nuevo llamado, pero esta vez la lista ira sin el primer dato; si la condición arroja falso, se retorna la lista ingresada, que en este caso será vacío.

Como resultado se debe de obtener:

Ilustración 7. Resultado de un programa en Dr Racket

Ejercicio No 3.

Hacer una función que reciba un documento de texto y elimine o cambie todas las letras “a”.

```
(define p (open-input-file "formato.txt"))

(define (f x)
  (define a (read-char p))
  (if(eof-object? x)
     (begin
       (close-input-port p)
       (list))
     (begin
       (if(char=? #\a a)
          (begin
          (cons x (f (read-char p))))
          (begin
          (if(equal? #\a x)
             (begin
               (cons a (f (read-char p))))
             (cons x (f a))
             )
          )
          )
       )
  )
  )

(f p)
```

Análisis.

P almacena el archivo de entrada “*formato.txt*”. La función *f* tendrá como argumento a *x* quien recibirá un fichero dado. En *f* se define *a*, variable que almacena el procedimiento de (*read-char p*), esto para guardar y cambiar el valor de cada dato en el archivo. Se determina si *x* es un *final de línea*; si es verdadero se cierra el archivo y se crea una lista con cada elemento que compone al fichero; si es falso se obtienen dos nuevas condiciones en

donde se iguala el valor de *a* con el carácter #\a; si esto se cumple, se crean parejas a partir del valor que tome *x* y el llamado recursivo de la función con argumento (*read-char p*),lo que permitirá mantener el orden de las oraciones eliminando las que se asemejen al carácter anterior y uniendo letra tras letra; siendo falso se evalúa si el valor tomado por x es igual al carácter buscado, por lo que, si esto es verdadero, se crean nuevamente parejas con el valor tomado por *a* y el nuevo llamado de la función con argumento (*read-char p*) generando el mismo procedimiento que en el bloque anterior, pero evaluando posiciones distintas (*a* siempre toma el carácter que le sigue a *x*); siendo falso, se forman parejas a partir de *x* y el llamado de la función que tendrá como argumento el valor almacenado en *a*.

Como resultado se debe de obtener:

```
>  (#<input-port:/Users/miguellopez/Desktop/LIBRO
   programacion/formato.txt>
    #\H #\o #\l #\space #\c #\o #\m #\o #\space #\e #\s #\t #\s
   #\space #\s #\e #\space
   #\l #\l #\m #\space #\A #\n #\g #\e #\l)
```

6.3 Ejercicio de repaso

Se proponen a continuación una serie de ejercicios para afianzar los conceptos vistos en esta sección.

1.A partir del ejercicio No. 3 expuesto en “Archivos”, realice un programa que reciba como parámetros el nombre del archivo, un carácter a buscar y otro el cual será el sustituto. El programa deberá de cambiar n todo el archivo el carácter buscado por su reemplazo.

2.Hacer una función que reciba un archivo y calcule el número de líneas que lo componen.

3.Hacer un programa que reciba un archivo con un texto almacenado. Las palabras que se encuentren en la posición par serán guardadas en un archivo llamado “par.txt”, de igual forma, las impares irán en un archivo “impares.txt”. El documento original elimina.

4.Hacer un programa que desencripte o reconstruya el mensaje a partir de los dos archivos generados, “par.txt” e “impar.txt”.

5.Hacer una función que reciba un archivo con texto. A cada posición que sea un numero primo, se le deberá de sumar 2 al valor del carácter. Al final deberá guardar el nuevo texto en un nuevo documento.

6.Hacer un programa que cuente el número de palabras que componen a un archivo.

7.A través de una estructura *persona*, crear un archivo que guarde 5 registros de la estructura, una línea por registro.

Servicios web con Dr Racket

A continuación, se mostrarán temas relacionados con la programación funcional en el desarrollo web en cuanto a teoría y creación de páginas de internet, servidores, gestión de bases de datos y seguridad de la información.

7. Servidores y comunicación a través de la web

7.1 Protocolo de control de transmisión (TCP)

Este procedimiento es fundamental para el internet en toda su expresión, gracias a este se es posible las conexiones virtuales. Este protocolo crea conexiones entre si para dar paso a un flujo de datos los cuales son entregados al destino sin cometer errores y en el orden en que fueron enviados. Se correlaciona con los puertos, diferenciando varios tipos de destino a los que se dirige la información dentro de un mismo sistema.

El protocolo TCP a través de una red interna o externa se comunica de forma segura con cada objeto que envié información, dicho envío es hecho a partir de múltiples subprocesos.

Otras características que cumple son:

- Al monitorear la llegada y envió de datos, regula el flujo de datos evitando así la saturación de la red.
- Permite multiplexar los datos, es decir, circular simultáneamente información de distintas fuentes.
- Iniciar y terminar un proceso de manera ordenada.

7.2 Funcionamiento del protocolo TCP

El TCP establece conexiones entre dos o más puntos terminales dentro de una red informática, en donde sea posible el intercambio de datos mutuo. Hace parte de la familia de protocolos de transporte con UDP (Protocolo de Datagramas de Usuario) y SCTP (Protocolo de control de transmisión de streaming). Con ellos se compone la arquitectura principal de comunicación dentro una red.

TCP permite mezclarse con muchos protocolos de internet (IP, RARP, IGMP, entre otros), de aquí la creación de distintos tipos de flujos de información (servidores, telefonía, etc.); esto se corrobora más adelante con el proceso de construcción de un servidor web a través del entorno de Dr Racket.

Construcción

Algunos conceptos para tener en cuenta son:

Socket: Mecanismo de entrada y salida que permite la comunicación entre procesos, es decir, intercambio de información.

Puerto: Sus definiciones varían según el campo en el que se aplique. En las conexiones de internet, los puertos de red se definen como un punto de destino al cual llegara una transferencia de datos. Estos son necesarios para mantener la comunicación entre dos partes dentro de una red virtual.

SYN: Establecen la conexión entre dos partes para el protocolo de control de transmisión.

ACK: Confirma la recepción de paquetes TCP, es decir, verifica el ingreso (o recepción) de información de una parte a otra.

FIN: "Finish" indica si alguna de las partes conectadas termina la transmisión de información.

PSH: "Push" entrega directamente los datos enviados de un emisor al receptor.

RST: "Reset" restablece la entrega de datos si ocurren errores de transmisión.

URG: "Urgent" prioriza el procesamiento de los datos particulares.

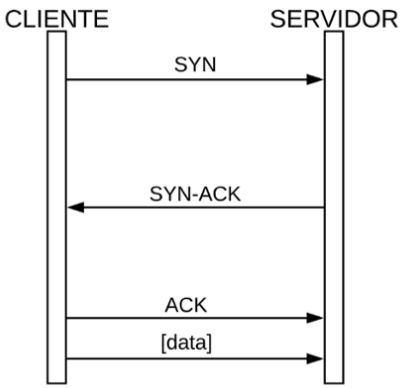


Ilustración 8. Protocolo de control de transmisión

Las conexiones TCP se ejecutan en tres procesos:

- Establecimiento de conexión (3-way handshake)
- Transferencia de datos
- Fin de la conexión.

Establecimiento de la conexión

Se tiene un punto A (servidor, etc.), este determina un puerto para recibir información y un socket para enviar; el punto queda a la escucha de nuevas conexiones. Luego de la espera, se conecta un punto B (cliente) al canal de información de A. B comienza a enviar unidades de transmisión (llamadas también paquetes), es decir, unidades SYN. Se debe tener en cuenta que B debe de estar igualmente configurado con puertos y sockets capaces de atender al flujo de entrada y salida para el lado A. Este último recibe la información SYN de B, y cómo respuesta devolverá unidades de tipo SYN/ACK; para terminar la fase de conexión, B envía unidades ACK hacia A, concluyendo la unión de los dos para el traspaso de datos.

Transferencia de datos

Se lleva a partir de varios procesos encargados de detectar errores, organizar información, monitorear llegadas o retrasos de información, entre otros. Algunos de esos procesos son:

Números iniciales de secuencia: Cada unidad o paquete que es enviado contiene una secuencia de datos que describen la información enviada. Esos conjuntos de

secuencias son leídas por las dos partes enlazadas para reconocer el tipo y cantidad de información enviada por paquete (la capacidad máxima de envío es de 1500 bytes).

Checksum: Analiza la veracidad de los números iniciales de secuencia. Detecta cambios ocurridos en la secuencia identificadora, con el fin de proteger la integridad de los datos, evitando disrupciones entre las solicitudes de envió para cada lado.

Ventanas deslizantes: Procedimiento que controla el flujo de datos entre el emisor y receptor. Evita que se envíen datos más rápido de lo que el receptor puede enviar y recibir. El control se lleva a través de la configuración del buffer y el número total de bits a almacenar.

Fin de la conexión: Llamada también "4-way handshake", puede terminarse la comunicación por ambas artes, sin embargo, también es posible que un lado (cliente o servidor) dejen de intercambiar información enviando como unidad de transmisión el elemento FIN, con el cual la parte opuesta del flujo de datos responderá con el paquete ACK. Similar a la transmisión usual anteriormente descrita, pero con un paquete de envío distinto.

7.3 Puertos TCP

El protocolo de control de transmisión utiliza puertos con ciertos números que distinguen el tipo de emisores y receptores a utilizar. Cada lado del flujo de información tiene asociado un numero de puerto (existen 65536 puertos posibles). Existen tres tipos de asignación:

- **Conocidos:** Nombrados por la IANA (Internet Assigned Numbers Authority). Van del 0 al 1023, y se asignan principalmente a sistemas. Algunos ejemplos son FTP (21), SSH (22), TELNET (23), SMTP (25) y HTTP (80).
- **Registrados:** Usados de forma temporal por aplicaciones. Van desde el 1024 hasta el 49151.
- **Dinámicos/privados:** Usado por aplicaciones que asignan dinámicamente los puertos al momento de establecer la conexión cliente-servidor. Van desde el 49.152 al 65.535.

Nota: en informática, el puerto 80 se conoce como el puerto por defecto por el cual un servidor HTTP recibe la petición hecha por un cliente.

7.4 Concepto sobre servidores

Del término general servicios, la palabra servidores se refiere a la unidad informática física o virtual que brinda múltiples servicios a equipos de cómputo que se encuentren conectados a una red. Puede ser un ordenador ya sea de hogar, o los que suelen verse en grandes empresas como Google, con habitaciones inmensas repletas de máquinas que parecen armarios, los cuales son simplemente computadoras. También puede ser un software que ayude a manipular correctamente los servicios de comunicación brindados por la maquina a un cierto grupo de equipos.

Un servidor es capaz de tomar peticiones de los equipos beneficiados y devolver una respuesta en concordancia, por ejemplo, cuando realizamos una búsqueda en Chrome lo que sucede es que hacemos una petición de búsqueda al

servidor que nos configura (Google), esta toma dichas ordenes, las analiza, y devuelve una respuesta que va de acuerdo con lo que se pidió, en este caso el resultado de la búsqueda. A este modelo se le denomina estructura **cliente-servidor**, donde el cliente compone a los ordenadores alojados en el servidor, y el servidor, que es el núcleo por el cual pasa cada acción ejercida en un ordenador.

Servicios como internet, web, telefonía, mensajería virtual, entre otros, basan sus funcionamientos en la estructura cliente-servidor, con la diferencia que cada uno adopta particularidades que se verán más adelante.

El modelo anterior se puede representar de la siguiente manera:

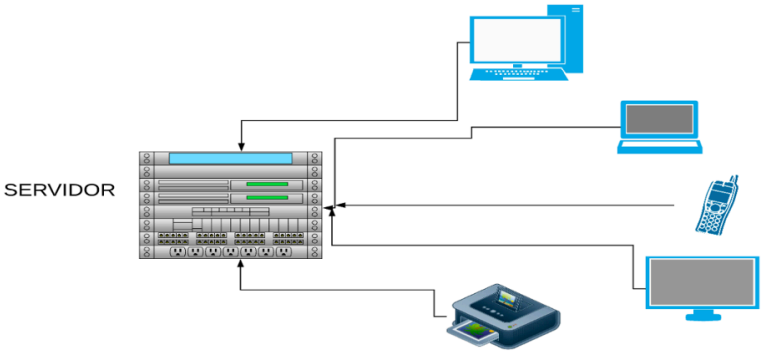


Ilustración 9. Funcionamiento de un servidor

Tipos de servidores

De acuerdo con el objetivo del servidor, estos se pueden clasificar en:

- **Servidor de archivos:** Llamado también *file server*, es el encargado de almacenar y distribuir archivos a todos los clientes.
- **Servidores de correo:** Recibe, almacena, envía y realiza otras operaciones relacionadas con correos electrónicos.
- **Servidores de bases de datos:** Gestionan el manejo de grandes volúmenes de información.
- **Servidores web:** Gracias a estos se pueden visualizar miles de páginas web actualmente. Permiten el almacenamiento de información HTML, texto, imágenes y demás material compuesto por diferentes datos.
- **Servidor de acceso remoto (RAS):** Controla las líneas de comunicación de la red y el cliente. A través de este se reconoce y autentica que usuarios se encuentran admitidos en una red particular.
- **Servidor proxy:** Monitorea el acceso a internet restringiendo el uso de distintos sitios web. También almacena información previamente adquirida de un cliente, con el fin de que en futuras ocasiones la conexión se haga mucho más rápido.

Existen más tipos de servidores, dependiendo del campo a tratar (telefonía, informática, etc.). De lo anterior, se obtienen algunos modelos de comunicación fundamentales para el desarrollo web, informática y telecomunicaciones, entre otras.

7.5 Ejemplo de un servidor web

Construcción

Para la construcción de servidores a partir del modelo TCP se deben conocer 3 funcionales principales:

- **Tcp-listen:** Abre un puerto en el ordenador para escuchar las conexiones que lleguen de cualquier sitio. Su estructura es:
  - `(tcp-listen <número de puerto> <número de conexiones> <reutilización del puerto> <opción / parametrización de direcciones>)`

Nota: Reutilización y parametrización son de tipo #f predeterminado. Si este último llegase a ser falso, las direcciones que aceptaría deben de estar definidas, como "127.0.0.1", lo que indica que solo se aceptaran conexiones guiadas a ese destino.

- **Tcp-accept:** Acepta las conexiones de un cliente. Su estructura es:
  - `(tcp-accept <receptor>)`

- **Tcp-connect:** Solicita una conexión con un servidor. Su estructura es:
  - `(tcp-connect <nombre de host> <número de Puerto> <nombre de host local> <número de puerto local>)`

Host: *Anfitrión, contraparte que provee servicios.*

- **Tcp-close:** Termina la escucha o conexión de un puerto. Su estructura es:
  - `(tcp-close <escuchante>)`

Construcción

Nota: Todo será explicado a partir de la documentación oficial de Dr Racket, por lo que para conocer secciones especificas es recomendable visitar el sitio web oficial.

El siguiente programa da pie a un servidor TCP/IP (Protocolo de control de transmisión/Protocolo de internet). La conexión se hace a través de un puerto IP, es decir, proveniente de una red que conecta el ordenador a la web. Se tiene:

```
1   #lang Dr Racket
2
3   (define (serve port-no)
4     (define listener (tcp-listen port-no 5 #t))
5     (define (loop)
6       (accept-and-handle listener)
7       (loop))
8     (loop))
9
10  (define (accept-and-handle listener)
11    (define-values (in out) (tcp-accept listener))
12    (handle in out)
13    (close-input-port in)
14    (close-output-port out))
15
16  (define (handle in out)
17
18    (display "HTTP/1.0 200 Okay\r\n" out)
19   (display  "Server:  k\r\nContent-Type:  text/html\r\n\r\n"
    out)
20  (display "<html><body>Hello, world!</body></html>" out))
```

Serve recibe como argumento el número de puerto. Para entablar la comunicación con un servidor se debe de utilizar el elemento *tcp-listen* (tendrá 5 conexiones disponibles). La comunicación deberá estar en constante movimiento, es decir, envío de peticiones y respuestas de tipo SYN y ACK, hasta que se termine el ciclo (paquetes de transmisión FIN), por lo que se decide manejar la función *loop*.

Accept-and-handler es usada en la función anterior. Esta ultima se encargará de recibir el lado que escuchará nuestra conexión, el servidor, usando tcp-accept; el elemento *define-values* propiciado por *#lang-Dr Racket* da valores correspondientes en el orden de escritura para *in* y *out,* cada uno recibiendo la comunicación entre las partes. Se llama a la función *handle* y los puertos *in* y *out* se cierran cuando *loop* finaliza.

Handle recibe puertos de entrada y salida y a través de estos despliega la información enviada; las dos primeras líneas envían las peticiones las instrucciones al servidor acerca del tipo de información a manejar; la ultima es un formato HTML que muestra "*hello, world!*" en pantalla.

Ejecución

En la consola de Dr Racket, se debe de llamar a la función *serve* con un numero de puerto; anteriormente se comentó los tipos de puertos, y en este caso, para comunicaciones HTTP se usa el puerto No. 8080.

Luego de hacer el llamado, se debe dirigir hacia algún navegador web y buscar en la zona de URL's la dirección *localhost: 8080*.Se debe visualizar:

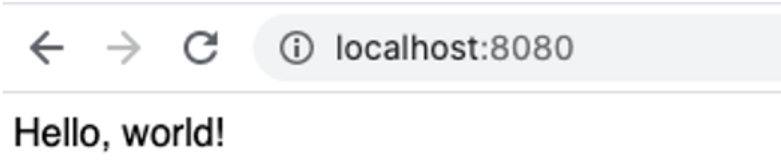


Ilustración 10. Servidor

El servidor está corriendo en una maquina local, es decir, en el ordenador que se esté usando. *Localhost* (host local) es la función que determina cómo se ejecuta un servicio en el navegador. Al ser local significa que la información mostrada es provista por el cliente y que solo podrá ser vista en la maquina en la que sea manejada.

8. Desarrollo de un aplicativo web

8.1 Servlets

Son programas que se ejecutan dentro de un servidor, generalmente de tipo web. Proviene del lenguaje Java como una ampliación de las capacidades de los servidores. El uso más común de los *servlets* es generar páginas web de forma dinámica a partir de los parámetros de petición que envíe el navegador web o el cliente.

Algunas de sus funciones particulares, en comparación los CGI (interfaces de entrada común, primeros modelos de comunicación dentro de la web) son el procesamiento de formularios o *formlets*, reenvío de peticiones a otros servidores y servlets, reducir el uso de memoria y persistencia (se mantienen activos luego de terminar la petición).

La ejecución de un servlet se muestra en la siguiente imagen:

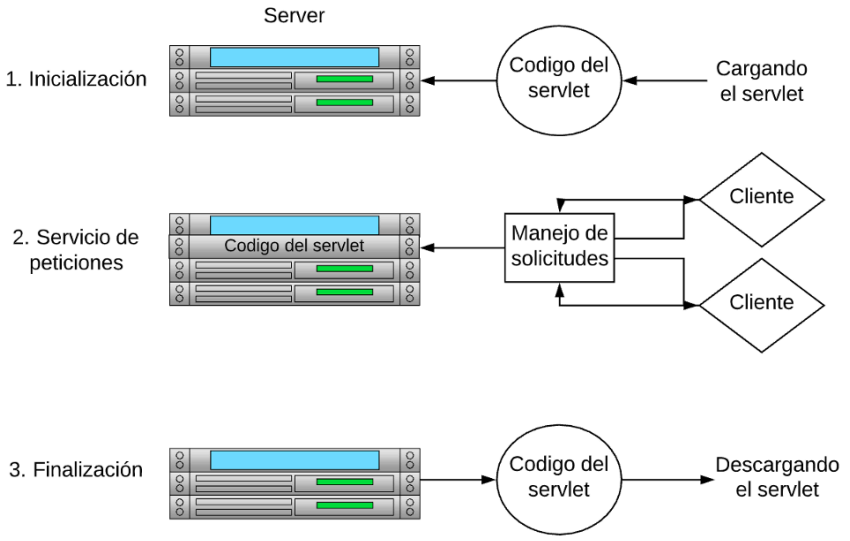


Ilustración 11. Ejecución de un Servlet

8.2 Creando un Blog en Dr Racket

En esta sección aprenderá a crear aplicaciones web dinámicas. Con ayuda de conceptos vistos en secciones anteriores, se explicará cómo iniciar un servidor web,

cómo generar contenido web interactuable con el usuario, entre muchas otras, a través de un blog.

Nota: Todo será explicado a partir de la documentación oficial de Dr Racket, por lo que para conocer secciones especificas es recomendable visitar el sitio web oficial.

Aspectos que se deben cumplir.

-Al ser un blog (principalmente publicaciones), se tendrá la capacidad de crear publicaciones y agregar comentarios.

-Se mostrarán en forma de lista las publicaciones creadas.

-Permitir comentar y compartir publicaciones.

-Modularizar toda la estructura del blog.

El blog

Un blog es un sitio web lleno de publicaciones y comentarios referentes. Estos serán llamados post; todo post se compone de un título y un cuerpo por lo que se debe pensar en algún procedimiento que permita almacenar múltiples campos (o variables). Pueden servir algunas estructuras de datos como los modelos *struct*.

Creando una estructura con los campos anteriores seria:

```
1 (struct post (title body))
```

A partir de esta forma se pueden crear algunos ejemplos bases como forma de guía cuando se ejecute el plano HTML, como los siguientes:

```
1  (define BLOG (post “Primer post!!” “Este es mi primer post!”))
```

Puede utilizar listas para almacenar distintos posts, como:

```
1  (define BLOG
2    (list (post "Second Post" "This is another post")
3          (post "First Post" "This is my first post")))
```

8.3 Desarrollo HTML

Toda página web se posiciona en un navegador usando su URL, por lo que, en temas ya vistos como servidores se sabe sobre la comunicación virtual a través de peticiones y respuestas. Cuando una persona busca en un navegador, este usa un servidor principal con el cual almacena toda la cantidad de información posible acerca de páginas relacionadas al tema de búsqueda. Cuando se clickea en alguna de todas las opciones el navegador genera una petición (siendo el cliente) al sitio web pidiendo que se despliegue toda la información que almacene esa dirección virtual. Como respuestas existen los formatos HTML, una forma común de escribir páginas web.

Para generar respuestas primero se debe tomar una solicitud de conexión o apertura:

```
(define (estar request) …)
```

Para cargar el código del blog, Dr Racket, a través de *response/xexpr*, función dada por la librería *web-server/insta*

proporciona funciones que toman sintaxis HTML y las devuelven como respuesta de la solicitud entrante.

```
1  (require web-server/insta)
2  (define (render-blog-page a-blog request)
3    (response/xexpr
4     `(html (head (title "My Blog"))
5            (body (h1 "My Blog")
6                  ,(render-posts a-blog)))))
```

Note que al final se llama una nueva función *render-posts* la cual se encarga de cargar cada publicación hecha, y en este caso se usaran las definiciones de BLOG hechas anteriormente.

El manejo de etiquetas HTML se hace teniendo en cuenta las sintaxis de listas y nombre de cada etiqueta, por ejemplo:

HTML
 este es un ejemplo

HTML
pasado

HTML
 Este es
otro ejemplo.

La jerarquía de los paréntesis dependerá de la representación natural de lo que se quiere hacer.

```
1  (define (render-greeting a-name)
2    (response/xexpr
3     `(html (head (title "Welcome"))
4            (body (p , (string-append "Hello " a-name))))))
```

Para ciertas instrucciones, se deben manejar modelos que combinan cadenas y texto en código, por ejemplo:

En estos casos Dr Racket provee una sintaxis llamada “cuasiquote”, que permite evaluar sub-expresiones rodeadas de comillas dobles, anteponiendo una coma al elemento mas externo que se quiere ejecutar.

Nota: consultar más sobre la extensión de sintaxis en “Como diseñar programas”.

Continuando, se crea la función *render-post*, que con ayuda de las estructuras de HTML se crean clases o divisiones (similares a las funciones) de tipo “post” en donde se despliega en pantalla toda la información que haga parte de la estructura inicial.

```
1  (define (render-post a-post)
2    `(div ((class "post"))
3          ,(post-title a-post)
4          (p ,(post-body a-post))))
```

Por ultimo se usa una función *post-body* la cual muestra cada publicación hecha:

```
1  (define (render-posts a-blog)
2    `(div ((class "posts"))
3          ,@(map render-post a-blog)))
```

Con ayuda de un iterador *map* se recorre cada post almacenado, iterando primero por *títulos* y luego por *cuerpo*. El @hace la acción de presentar organizadamente una lista en formato HTML (esto proveniente de Cuasiquote y expresiones x, de xepr).

Retomando.

Uniendo cada programa hecho en estas secciones, se forma lo siguiente:

```
1   (require web-server/insta)
2   ; A blog is a (listof post)
3   ; and a post is a (post title body)
4   (struct post (title body))
5
6   ; BLOG: blog
7   ; The static blog.
8   (define BLOG
9     (list (post "Second Post" "This is another post")
10          (post "First Post" "This is my first post")))
11
12  ; start: request -> response
13  ; Consumes a request, and produces a page that displays all
    of the
14  ; web content.
15  (define (start request)
16    (render-blog-page BLOG request))
17
18  ; render-blog-page: blog request -> response
19  ; Consumes a blog and a request, and produces an HTML page
20  ; of the content of the blog.
21  (define (render-blog-page a-blog request)
22    (response/xexpr
23     `(html (head (title "My Blog"))
24            (body (h1 "My Blog")
25                  ,(render-posts a-blog)))))
26
27  ; render-post: post -> xexpr
28  ; Consumes a post, produces an xexpr fragment of the post.
29  (define (render-post a-post)
30    `(div ((class "post"))
31          ,(post-title a-post)
32          (p ,(post-body a-post))))
33
34  ; render-posts: blog -> xexpr
35  ; Consumes a blog, produces an xexpr fragment
36  ; of all its posts.
37  (define (render-posts a-blog)
38    `(div ((class "posts"))
39          ,@(map render-post a-blog)))
```

Al ejecutar estas lineas en el entorno Dr Racket debera ver lo siguiente:

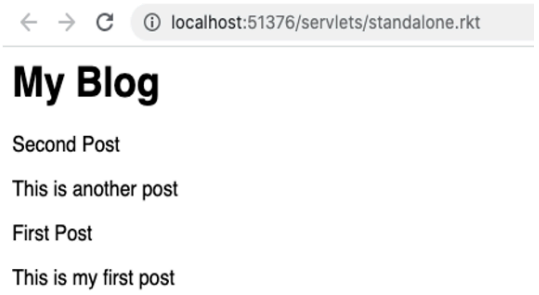


Ilustración 12. Planos HTML

Envió de publicaciones

Hasta ahora se tiene un blog estático donde cada publicación esta predefinida. Es momento de dinamizar su comportamiento, por lo que se agregara una función que permita subir nuevas publicaciones al momento de darle al botón enviar. Debe tener en cuenta que, al momento de cargar nueva información, se producen nuevas solicitudes, por lo que se pueden usar para verificar de donde provienen, facilitando el acceso a la estructura *post* para su futuro despliegue en pantalla.

Para el manejo de las nuevas solicitudes y su relación con el cuerpo de una publicación se tienen las siguientes funciones:

Request-bindings: Extrae toda la información de la solicitud hecha al cargar la nueva información.

Extract-binding/single: Extrae información específica de la solicitud realizada. Con esto se podrá conocer la nueva publicación.

Exists-binding?: Comprueba que un conjunto de solicitudes contiene un campo especifico.

De lo anterior, se tiene lo siguiente:

```
1  (define (can-parse-post? bindings)
2    (and (exists-binding? 'title bindings)
3         (exists-binding? 'body bindings)))
```

Esta definición can-parse-post? (“puede analizar el post” en español) es de tipo booleano, verifica que las solicitudes (llamarlo también enlaces o conexiones) hacen parte de la estructura post, reconociendo enlaces para el campo *title* y *body*. Si es verdadero se muestra la publicación en pantalla; si es falso, se retorna vacío.

Se deben de comprobar si la solicitud de carga contiene los campos necesarios para publicar un nuevo post, por lo que se crea lo siguiente:

```
1  (define (parse-post bindings)
2    (post (extract-binding/single 'title bindings)
3          (extract-binding/single 'body bindings)))
```

Parse-post (“analizar el post” en español) extrae la información de la solicitud, tomando cada campo por independiente y a través del modelo definido *post*. Con esto se obtiene el título y el cuerpo de lo que el usuario ingreso.

Todos los fragmentos mostrados hacen parte del campo de solicitudes, sin embargo, para generar dichas solicitudes debemos organizar un proceso al final de la página, el cual permita escribir un mensaje y a través de un botón cargarlo

en la página, a manera de listas, ubicándolos en orden de llegada.

```
1   (define (render-blog-page a-blog request)
2     (response/xexpr
3      `(html (head (title "My Blog"))
4             (body
5              (h1 "My Blog")
6              ,(render-posts a-blog)
7              (form
8               (input ((name "title")))
9               (input ((name "body")))
10              (input ((type "submit")))))))))
```

Usando la estructura del lenguaje HTML se crean dos casillas para *title* y *body*, donde se compondrá la publicación. Por último, un botón *submit* que dará la orden de envío de la solicitud de carga.

Recopilando todos los fragmentos, el programa debe mostrar lo siguiente:

Ilustración 13. Publicaciones

8.4 Control de peticiones

Si se analiza por un momento el código, encontrará que las conexiones principales como el despliegue del blog por

parte del navegador y la publicación de nuevos posts dependen de la función *start* o *inicio*. Esto limita el comportamiento del código, sin embargo, es posible re-direccionar las solicitudes que ingresen a distintas funciones gracias a la biblioteca de *web-server,* que provee la función *send/suspend/dispatch*.

Continuando, se juntan todas las necesidades y se reforma la estructura de *render-blog-page* para tener lo siguiente:

```
1   (define (render-blog-page a-blog request)
2     (define (response-generator embed/url)
3       (response/xexpr
4        `(html (head (title "My Blog"))
5               (body
6                (h1 "My Blog")
7                ,(render-posts a-blog)
8                (form ((action
9                        ,(embed/url insert-post-handler)))
10                      (input ((name "title")))
11                      (input ((name "body")))
12                      (input ((type "submit")))))))))
13
14    (define (insert-post-handler request)
15      (render-blog-page
16       (cons (parse-post (request-bindings request))
17             a-blog)
18       request))
19    (send/suspend/dispatch response-generator))
```

El cuerpo HTML se almacena en *response-generator* (“generador de respuesta” en español) que tiene como argumento una función de *web-server*, *embed-url*, la cual reinserta la URL del blog con la acción que se haya ejecutado, en otras palabras, con este pedazo de código se transfieren los datos en términos de URL.

Insert-post-handler (“insertar el manejador de publicaciones” en español) da un nuevo sentido a cada solicitud entrante. Cuando se hace un nuevo llamado, se

crean pares a partir de las nuevas solicitudes creadas por la carga de nuevas publicaciones y el blog actual (siendo BLOG definición en uso) y la petición *request* , que es la misma petición generada por los nuevos datos. Al final se utiliza la función que separa la solicitud inicial en distintas partes.

Nota: Observe que el código se estructura de una manera poco usual al formato que se ha venido trabajando, usando funciones antes de su definición.

8.5 Igualdad de forma

El blog actual presenta un gran problema, para descubrirlo, ejecute el programa y cargue un nuevo post, mientras eso, busca en una nueva ventana la misma URL del blog. Lo que sucederá es que los cambios no se guardaran y mostrara la versión escrita en código que se usa como referencia. Es aquí donde se plantea las posibilidades de reforma que permitan hacer de esta página un blog en el que se comparta toda la información.

El blog se estructuró en modelos de tipo *struct*, por lo que se está en la capacidad de agregar propiedades de mutación a las estructuras formadas, permitiendo cambiar la información almacenada en tiempo constante, por ejemplo:

```
1 (struct blog (posts) #:mutable)
```

La propiedad *mutable* también permite cambiar la

```
1  (define (blog-insert-post! a-blog a-post)
2    (set-blog-posts! a-blog
3              (cons a-post (blog-posts a-blog))))
```

información antes de almacenarla en el servidor gracias a funciones de campo como *set-estructura-campo*:

Blog-insert-post! se encarga de *posicionar* las nuevas publicaciones al inicio de la página. Usa *set-blog-posts!* para cambiar el comentario actual por la nueva información cargada, usando parejas entre el post actual y las nuevas solicitudes.

Al realizar cambios en la carga de nuevas publicaciones, se debe también modificar la función *insert-post-handler* para que las nuevas peticiones se realicen de acuerdo con el criterio de cambio de *struct blog* :

```
1  (define (insert-post-handler request)
2      (blog-insert-post!
3       BLOG(parse-post (request-bindings request)))
4      (render-blog-page request))
```

Se mantiene el proceso cada vez que se genera una solicitud de carga, se cambia la URL, se usa la nueva función de manejo de publicaciones con argumentos de BLOG (como blog referente) y el análisis de solicitudes para estructuras *posts*. Al final se itera en la llamada a la función principal de *render...,* cargando la nueva información.

Comentarios en las publicaciones.

Los comentarios pueden implementarse de múltiples formas, por ejemplo una estructura independiente, sin

embargo, para facilitar las cosas se creará un campo *comments* en la estructura *post*, allí se almacenarán todos los comentarios de cada publicación realizada.

Pueden ser gran cantidad de comentarios que se almacenen por publicación, por lo que se deben de cargar en número y por orden de llegada. Se pueden usar las listas ya que permiten incrementar su tamaño a medida que más cambios surgen en este campo. Se debe tener en cuenta que los datos irán cambiando para cada publicación, por lo que, como se realizó en la sección anterior, la estructura de *post* debe de tener la propiedad *mutable*.

Teniendo en cuenta lo anterior se tiene:

```
1  (struct post (title body comments) #:mutable)
```

Con ayuda de la siguiente función:

```
1  (define (post-insert-comment! a-post a-comment)
2    (set-post-comments!
3     a-post
4     (append (post-comments a-post) (list a-comment))))
```

Post-insert-comment! cambiará el comentario vigente en el campo de *post* por una lista hecha de la unión de información antigua en el campo *comments* y la ingresada por el formato de referencia BLOG, generando así una lista descendente.

Se deben de hacer otros ajustes como indexar hacia una nueva pantalla la publicación con sus comentarios, el manejo de las peticiones por parte de los comentarios, el

mapeo de cada comentario para presentarlos en manera de lista, entre otros.

Iniciando con el plano HTML, se tiene:

```
1   (define (render-post-detail-page a-post request)
2     (define (response-generator embed/url)
3       (response/xexpr
4        `(html (head (title "Post Details"))
5               (body
6                (h1 "Post Details")
7                (h2 ,(post-title a-post))
8                (p ,(post-body a-post))
9                ,(render-as-itemized-list
10                 (post-comments a-post))
11               (form ((action
12                 ,(embed/url insert-comment-handler)))
13                     (input ((name "comment")))
14                     (input ((type "submit")))))))))
15
16    (define (parse-comment bindings)
17      (extract-binding/single 'comment bindings))
18
19    (define (insert-comment-handler a-request)
20      (post-insert-comment!
21     a-post (parse-comment (request-bindings a-request)))
22      (render-post-detail-page a-post a-request))
23    (send/suspend/dispatch response-generator))
```

Se crea una función *render-post-detail-page* que recibe un post y una petición, se hace el cambio de URL con *embed/url* y se describen las partes que componen la nueva pantalla en donde se mostraran los comentarios. Posteriormente se hace la llamada a una función para listas *render...,* de la cual se hablará en unos momentos . Al final se crean los campos y botones para manejar los comentarios.

Parse-comment analiza las nuevas solicitudes para los comentarios y extrae la información de cada una para posteriormente cargarlos en la página. De igual forma lo maneja *insert...,* usando la función de insertar comentarios

a partir del post vigente y la información extraída por la definición anterior.

Observando toda la estructura que se ha creado, se tiene un blog que responde correctamente a nuevas publicaciones, y ahora con apartado de comentarios en una página detallada. Existe un inconveniente y es el cómo relacionar la pestaña de comentarios con la pestaña principal del blog, es decir, donde se sitúa cada post. Una manera útil es hiper-linkeando cada publicación, es decir, cuando el nombre del post reciba un click se redireccione a una pantalla con toda la información del post clickeado.

Para complementar estas nuevas acciones se puede agregar a la pestaña principal el número de comentarios por cada publicación, de esta manera el usuario podrá tener información más resumida sobre esa publicación.

Al querer hiper-linkear cada nombre para los posts se es necesario configurar las funciones de *render-post y render-posts*, como se muestra a continuación:

```
(define (render-post a-post embed/url)
  (define (view-post-handler request)
    (render-post-detail-page a-post request))
  `(div ((class "post"))
        (a ((href ,(embed/url view-post-handler)))
           ,(post-title a-post))
        (p ,(post-body a-post))
  (div ,(number->string (length (post-comments a-post)))
               " comment(s)")))
```

Observe que ahora la nueva función recibe como argumento un cambio de URL. Dentro de esta se crean funciones locales como *view-post-handler*, encargada de renderizar el nuevo título con el enlace hacia la página de

comentarios; con estructuras HTML la función *href* crea el hiperlink (o hipervínculo, de esta manera "ejemplo"). Al final se crea una nueva división para los comentarios donde convierten el número de comentarios existente en lista a una cadena, esto con el fin de evitar errores de peticiones, y en ciertos casos facilitar la funcionalidad de los procesos.

```
1  (define (render-posts embed/url)
2    (define (render-post/embed/url a-post)
3      (render-post a-post embed/url))
4    `(div ((class "posts"))
5          ,@(map render-post/embed/url (blog-posts BLOG))))
```

Render-posts crea una nueva URL a partir de cada publicación. Al final se mapea la clase *posts* para mostrar en pantalla todas las publicaciones existentes. Recordar en este caso la función del @, como agente organizador de todos los posts en listas.

```
1  (define (render-as-itemized-list fragments)
2    `(ul ,@(map render-as-item fragments)))
3
4  (define (render-as-item a-fragment)
5    `(li ,a-fragment))
```

Estas últimas funciones se encargan de ordenar todos los comentarios en una lista dada por el orden de llegada; *render-as-itemize-list* coge cada fragmento de lista (hecho por *render-as-item*) y con la etiqueta *ul* (unordered list en HTML) crea una nueva lista en la página de comentarios.

Retomando

Con todas las nuevas funciones, juntándolas en el documento que se esté trabajando, se debe de tener lo siguiente:

Ilustración 14. Publicaciones con hiperlink

Ilustración 15. Nueva publicación.

Ilustración 16. Página de comentarios

Arreglos de estilo.

Ahora que el blog se ejecuta correctamente es posible añadir nuevas funciones que hagan más cómodo la interacción del usuario, como botones para la movilidad entre las pestañas, condicionar la subida de comentarios, entre otras.

Movilidad

Se cuenta con dos secciones, la ventana principal, donde cada post es publicado con toda la información (título, cuerpo y comentarios), y la ventana de comentarios. Es algo tedioso utilizar la flecha izquierda del navegador para retroceder en un sitio web, por lo que se implementará una función capaz de redirigir al usuario a la pestaña principal.

Con lo anterior se puede crear lo siguiente:

```
1  (a ((href ,(embed/url back-handler)))
2                  "Back to the blog"))))))
3
4  (define (back-handler request)
5      (render-blog-page request))
6    (send/suspend/dispatch response-generator)
```

Con la etiqueta *href* y el cambio de URL se puede redirigir la dirección del blog a la pestaña principal solo cuando se clickea la zona de “Backs to the blog”.

Observe que *back-handler* renderiza nuevamente el blog haciendo un llamado a la función principal *render-blog-page*, creada en la sección 1.4.

Ejercicio: *Se tienen las funciones encargadas de redirigir al usuario desde la pestaña de comentarios a la pestaña de posts. Determine en qué lugar deben de ir para el correcto funcionamiento del blog.*

Confirmaciones

Existe dos posibilidades, la primera es estar de acuerdo con el comentario y publicarlo, la segunda es cambiar de parecer y evitar subirlo a la página. Para esto es necesario crear funciones que verifiquen cuál será la opción para elegir, un “si” o un “no” clickeables que después direccionen nuevamente a los comentarios.

Con lo anterior se tiene lo siguiente:

El funcionamiento es igual a las funciones que encargadas de publicar nueva información; *render-confirm-add-comment-page* recibe un comentario, un post y una petición. Se crea una nueva pestaña en donde se puede observar el comentario a subir y las opciones de “*yes, add the comment*” o “*No, i changed my mind*”, las cuales son hiperlinkeadas y cada una redirige a la pestaña de comentarios.

```
1   (define (render-confirm-add-comment-page a-comment a-post
    request)
2     (define (response-generator embed/url)
3       (response/xexpr
4        `(html (head (title "Add a Comment"))
5               (body
6                (h1 "Add a Comment")
7                "The comment: " (div (p ,a-comment))
8                "will be added to "
9                (div ,(post-title a-post))
10
11               (p (a ((href ,(embed/url yes-handler)))
12                     "Yes, add the comment."))
13               (p (a ((href ,(embed/url cancel-handler)))
14                     "No, I changed my mind!"))))))
15
16    (define (yes-handler request)
17      (post-insert-comment! a-post a-comment)
18      (render-post-detail-page a-post request))
19
20    (define (cancel-handler request)
21      (render-post-detail-page a-post request))
22
23    (send/suspend/dispatch response-generator))
```

Las funciones *yes-handler* y *cancel-handler* manejan la solicitud enviada al clickear en alguna de las opciones anteriores a través del llamado de *post-insert-comment* para cargar el comentario, y *render-post-detail-page* para mostrar la pestaña anterior.

Ejercicio: Determinar en qué sitio deben de ir cada una de las funciones anteriormente definidas.

Como resultado de lo anterior se debe tener:

Ilustración 17. Regreso de pestaña

Ilustración 18. Añadir un comentario

8.6 Solucionando algunos errores

Existe un fallo muy sutil en la carga de información. Estando en esta sección, al correr el actual programa del blog y se inserte un nuevo post, al recargar la página una y otra vez, lo que se verá es que se duplica la última información subida. Este error es conocido como “doble envió” y sucede a raíz de la mutación de estructuras por parte de las solicitudes realizadas. Al manejar funciones encargadas de cambiar cierta información como lo hace

yes-handler, las peticiones actuales son cambiadas junto a la estructura, y tales cambios se almacenan gracias al funcionamiento del blog.

Una posible solución es utilizar la función *redirect/get* para enviar las solicitudes de mutación a una URL distinta, en la que se pueda recargar. La implementación es:

```
1  (define (yes-handler request)
2      (post-insert-comment! a-post a-comment)
3      (render-post-detail-page a-post (redirect/get)))
```

Observe que al llamar la función *render-post-detail-page*, el argumento de petición es *redirect-get*, la solución al error de doble carga. Este enviara las nuevas solicitudes de cambio a una URL que acepte cada la información generada.

Divisiones

Es una buena práctica el manejar código por separaciones, donde cada parte prioriza una acción en particular. En Dr Racket se permite distribuir el código en distintos archivos y a través de comandos, hacerlos accesibles a cualquier documento que lo requiera. Esta metodología permite modificar el código desde distintas partes sin afectar el funcionamiento principal.

Ejercicio.

Determinar qué parte del código actual del blog hace referencia al modelo base representativo, en otras palabras, que funciones son el modelo de creación del blog. Crear un nuevo documento en donde guardar el código modelo, el

resto del programa lo dejara en el documento que ha venido trabajando.

Debe de utilizar las funciones *(provide (all-defined-out))* y *(require "nombre.rkt")* para poder hacer el documento modelo accesible a otros archivos.

8.7 Final

Siguiendo todos los pasos anteriores, se debe tener una página web (blog) funcional con capacidades para recibir y subir publicaciones a un servidor web local, asi como de comentar y asegurar las publicaciones a realizar.

Se recomienda consultar la documentación de Dr Racket para ampliar la información sobre creación de servlets, manipulación de servicios web, entre otras., para así lograr un mejor desempeño en el área de programación y desarrollo web.

9. Bases de datos

9.1 Conceptos

Una base de datos es un almacén de información que pertenece a un mismo contexto y se organiza de tal manera que permite su posterior uso, análisis y transmisión. Una biblioteca o un directorio de telefonía funcionan como una base de datos física, en donde toda persona tiene acceso a la información. Actualmente, para el desarrollo de la

tecnología se usan componentes virtuales para el manejo de la información, lo que hoy se conoce como DBMS (Database Management System o sistema de gestión de bases de datos). Los sistemas virtuales permiten almacenar y manejar rápida y estructuradamente grandes volúmenes de información.

Tipos de bases de datos

Existen múltiples tipos, como:

•Según su variabilidad en la forma de recuperar y preservar los datos. Existen dos tipos:

•**Estáticas:** Utilizados en la inteligencia empresarial, son bases de datos que solo admiten lectura de información. No se permite generar cambios en la información.
•**Dinámicas:** Manejan operaciones de consulta, actualización, borrado y edición de información.

•Según el modelo de organización existen:

•**Jerárquicas:** Su funcionamiento se basa en arboles binarios. El nodo padre o la raíz que contiene la información despliega hijos con datos subyacentes.
•**Red:** Se asimila con la estructura jerárquica de los arboles binarios, solo que las diferencias residen en que esta permite la apropiación de varios padres por parte de un nodo. Estas estructuras manejan los problemas de redundancia de datos, volviendo más eficiente y complicado el manejo de la información por lo que esta se guía fuertemente hacia los programadores.

- **Transaccionales:** Su objetivo es el envío y recepción de información a la mayor velocidad posible. Suelen utilizarse en la industria con procesos de manufactura y producción.
- **Relacionales:** Se basa en la idea de filas (registros) y columnas (campos) y las relaciones que pueden existir de estos. El lenguaje más habitual para este modelo es SQL (Structured Query Language o Estructurado de consultas).
- **Multidimensionales:** Utiliza un vector multidimensional para el almacenamiento de datos. Puede usar bases relacionales atribuyendo a los campos más de dos tipos, bien sea una dimensión (almacenaje de información general) o un parámetro de información (datos específicos).
- **Bases de datos orientadas a objetos:** Almacena toda la información referente al paradigma POO (programación orientada a objetos).

9.2 Bases de datos en Dr Racket

El entorno Dr Racket permite la interacción con los siguientes servicios de bases de datos:

- PostgreSQL
- MySQL
- SQLite
- Cassandra
- ODBC

Todas las opciones se basan en lenguaje SQL (System Query Language) para el tratamiento de datos.

SQL

Es un lenguaje de programación usado en la administración de sistemas de bases de datos relacionales. A través de operaciones algebraicas y calculo relacional, crea métodos de inserción, consulta, actualización, borrado, creación y modificación de información.

Funcionamiento

Para la implementación del lenguaje de consultas en Dr Racket se utiliza la sintaxis nativa de SQL a través de declaraciones por medio de funciones. A continuación se mostrará la sintaxis de consulta utilizada por cada servicio:

Nota: Para el uso de las funciones a continuación, se debe de instalar el paquete de bases de datos "db". Al final deberá requerir todas las funciones con la función ***(require db).***

Manipulación de datos (sintaxis SQL)		
Función	**Ejemplo**	**Descripción**
`select`	`"select <campo(s)> from <nombre de la tabla> …"`	Muestra información sobre los datos almacenados. Puede ser especifico o general.
`insert`	`"insert into <nombre de la tabla> values <campos>"`	Inserta datos en una tabla. Su uso se asemeja con el proceso de llamado a funciones.

Manipulación de datos (sintaxis SQL)		
update	“update <nombre de la tabla> set <valores nuevos> where <ruta para el cambio de datos>”	Actualiza la información de una tabla.
delete	“delete from <nombre de la tabla> where <ruta para eliminar un registro>”	Borra un registro de una tabla.
drop table	“drop table <nombre de la tabla>”	Elimina una tabla.
truncate table	“truncate table <nombre de la tabla>”	Elimina solo los datos de una tabla.

Definición de datos		
créate database	“créate database <nombre de la base de datos>”	Crea una base de datos.
use	“use <nombre de la base de datos>”	Inicia la base de datos.
create table	“create table <nombre de la tabla> (<definición de campos>)”	Crea una tabla.

Comandos adicionales para las funcionales SQL		
distinct	“select distinct <campo> from <nombre de la tabla>”	Muestra todos los datos sin sus repeticiones (si existen).
where	“select <campo> from <nombre de la tabla> where <condición>”	Crea condiciones para la muestra de datos.

Comandos adicionales para las funcionales SQL		
and/or	“select <campo> from <nombre de la tabla> where <condición> and/ or <condiciones>”	Propone operaciones lógicas de “y” y “o”. pueden usarse por separado de la siguiente manera: • and evalúa dos condicione verdaderas. • or evalúa una condición verdadera.
in	“select <campo> from <nombre de la tabla> where <campo> in <valores>”	Filtra la información mediante valores propuestos.
between	“select <campo> from <nombre de la tabla> where <condición> between <valor> and <valor>”	Retorna los datos de acuerdo a un intervalo entre datos.
like	“select <campo> from <nombre de la tabla> where <condición> like <coincidencia o patrón>”	Retorna todos los datos que contengan el patrón o coincidencia propuesta.
order by	“select <campo> from <nombre de la tabla> where <condición> order by <campo> <asc, desc>”	Ordena los elementos tomados de forma ascendente o descendente.
count	“select count <campo> from <nombre de la tabla>”	Cuenta el número de filas.
group by	“select count <campo> from <nombre de la tabla> group by <campo>”	Realiza agrupaciones.

Comandos adicionales para las funcionales SQL		
having	“select <campo> from <nombre de la tabla>… having <condición>”	Crea condiciones para la muestra de datos.

Funciones propias de Dr Racket		
query-exec	(query-exec <conexión de la base de datos> <declaración>)	Realiza una declaración de consulta en la base de tipo SQL. Esta consulta se realiza en el entorno que provea interfaz de bases de datos (MySQLWorkbench que será utilizado más adelante)
query	(query <conexión de la base de datos> <declaración>)	Como query-exec, pero retornando en la consola de Dr Racket el resultado de la consulta con la ruta de obtención.
query-rows query-row	(query-rows <conexión de la base de datos> <declaración>)	Devuelve una o varias filas en forma de vectores.
query-list	(query-list <conexión a la base de datos> <declaración>)	Retorna una lista de valores de la consulta realizada.
query-value	(query-value <conexión a la base de datos> <declaración>)	Retorna un resultado en específico de una sola fila y una sola columna.
in-query	(in-query <conexión a la base de datos> <declaración>)	Produce una sentencia de columnas de cierta fila, con sus valores respectivos de cada casilla.

Funciones propias de Dr Racket		
list-tables	(list-tables <conexión a la base de datos> <#:schema>)	Muestra información de la base de datos como una lista del número de tablas creadas. Los esquemas pueden ser: ● 'search-or-current ● 'search ● 'current
table-exists?	(table-exists <conexión a la base de datos> <nombre de la base de datos> <#:schema>)	Indica si una tabla existe. Los esquemas usados son los mismos que en list-tables
sql-null?	(sql-null? <elemento>)	Función boooleana que evalúa si x es un elemento nulo.
sql-date	(struct sql-date (<año> <mes> <fecha>)) <year>: número entero. <month>: número entero de 0 a 12. <day>: número entero de 0 a 31.	Representa la fecha para datos en SQL.
sql-time	(struct sql-time (<horas> <minutos> <segundos> <nanosegundos> <zon horaria>)) -Se utilizan números enteros no negativos	Representa el tiempo para datos en SQL.

Tabla 6. Funciones para el manejo de bases de datos

Nota: los puntos suspensivos manejados en algunas funciones indican información extra que puede resultar necesaria para un mejor desempeño de dicha función.

Los tipos de datos manejados en un lenguaje de consultas (SQL) son:

- Enteros (integer)
- Flotantes (floats)
- Char (caracteres)
- Fechas (date)

9.3 Desarrollo e implementación de una base de datos

Las bases de datos en Dr Racket pueden ser manejadas a través de conexiones TCP utilizando servicios anteriormente mencionados (MySQL, postgreSQL, Cassandra, entre otos.). Para mostrar el correcto manejo del lenguaje SQL y las bases de datos en el ambiente de Dr Racket, se usará el servicio de MySQL para obtener un servidor local dedicado al almacenamiento y tratamiento de datos.

Instalación y preparación de herramientas

Antes de iniciar, es necesario instalar MySQL Workbench y XAMPP.MySQL Workbench servirá como interfaz principal de manejo directo con la base de datos a crear; para el correcto funcionamiento se necesita correr una máquina virtual que simule ser el servidor que alojará la base de datos, en este caso XAMP mantendrá activa una IP local.

Para este ejemplo se utilizaron las siguientes versiones:

MySQL Workbench versión 6.3.10.
XAMP versión 7.4.2.

Para utilizar modelos actuales de cada aplicativo, es necesario conocer las nuevas implementaciones de la librería *db* para nuevas versiones de cada servicio, para ello consulte la documentación oficial de Dr Racket.

Inicio de servidor

Con los siguientes pasos:

- Iniciar XAMPP.
- Inicie el servidor local presionando en el botón de *start*.
- Luego de estar en verde la opción de *status,* dirigirse a la pestaña de *services*.
- Escoger MySQL y presionar el botón *start*.
- Mantener la aplicación abierta en todo momento.

Creación de base de datos

Con los siguientes pasos:

- Iniciar MySQL Workbench.

- Junto al título *MySQL* conncections presionar el símbolo 

- Obtendrá la siguiente ventana:

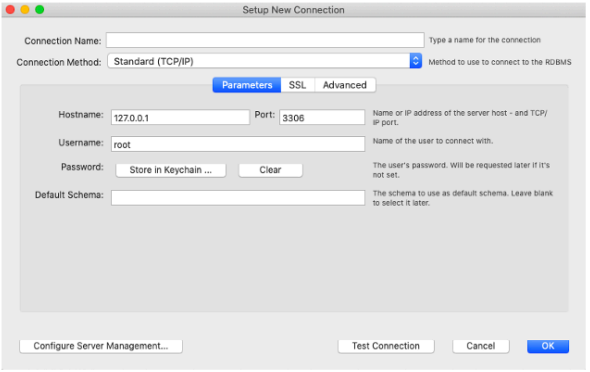


Ilustración 19. Creación de una base de datos

•Nombrar la conexión de la base de datos (para el ejemplo se usó el nombre de local).

•En el campo de *hostname* se pondrá la IP que proporciona XAMPP:

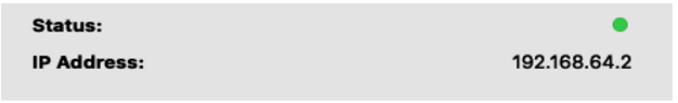


Ilustración 20. Dirección IP XAMPP para la base de datos

•Presionar el botón OK.

Nota: No se comentará el uso de la plataforma workbench, por lo que se recomienda consultarlo.

Primeros pasos

Hay que tener en cuenta el procedimiento de conexión a una base de datos MySQL, por lo tanto, se estructura de la siguiente manera:

- `(mysql-connect`

#:user <cel nombre de usuario con el que se creara la base de datos. Por defecto suele ser root>
#:database <cadena; nombre de la base de datos>
#:server <dirección del servidor donde se aloja la base de datos>
#:port <número de puerto>
#:socket <dirección del socket;en algunos casos se puede manejar el elemento (mysql-guess-socket-path) para encontrar cual es la ruta correcta para el flujo de información>
#:allow-cleartext-password?
<autenticación de información; por defecto se usa 'local>
#:ssl <(Secure Sockets Layers), otro modelo de autenticación, encriptación y descifrado de información enviada por internet; se puede elegir este protocolo con 'yes, 'optional o 'no>
#:password <contraseña con la que se crea la base de datos; por defecto suele ser root>
#:notice-handler <notifica sobre errores en la conexión; usa 'error o 'output>
)

Teniendo la definición de conexión MySQL, es momento de pasar al código. Luego de la creación de la conexión de la base de datos, esta arrojara la información necesaria para una conexión con otros programas, por lo tanto, aplicando la estructura se tiene:

Nota: Para manejar toda la sintaxis de consulta se debe de usar la siguiente estructura:

(<función> <conexión> <" sintaxis nativa SQL">)*

**Cada declaración debe ser escrita entre comillas.*

```
1  #lang Dr Racket
2
3  (require db)
4  (require sql db)
5
6  (define pgc (mysql-connect #:server "192.168.64.2"
7                  #:port 3306
8                  #:user "root"
9                  #:ssl 'no))
```

La estructura de conexión se almacena en una función *pgc* para el uso de funciones SQL. El *server*, *puerto*, *user* y opción *ssl* fueron proporcionados por las opciones de servidor de la interfaz de workbench.

A medida que se desarrolla todo el ejemplo las funciones de consultas hacia el entorno de MySQL se realizaran usando el campo de comandos de Dr Rakcet. Para comenzar la conexión entre las dos aplicaciones se debe de llamar a la función *pgc* que retornara lo siguiente:

```
>  pgc
   (object:connection% ...)
```

(object:connection% ...) indica que la conexión se ha realizado.

Creación de la base de datos y tablas.

Continuando, para crear la primera base de datos se debe usar lo siguiente:

```
>  (query- exec pgc "use empresa")
```

La función *query-exec* crea una consulta en el workbench usando como argumento la función que almacena la conexión, esto para indicar hacia donde se dirige la acción. La parte "*créate database empresa*" (en español "crear una base de datos empresa") crea la base de datos con nombre "empresa". Para confirmar su existencia se debe observar lo siguiente en la interfaz de workbench:

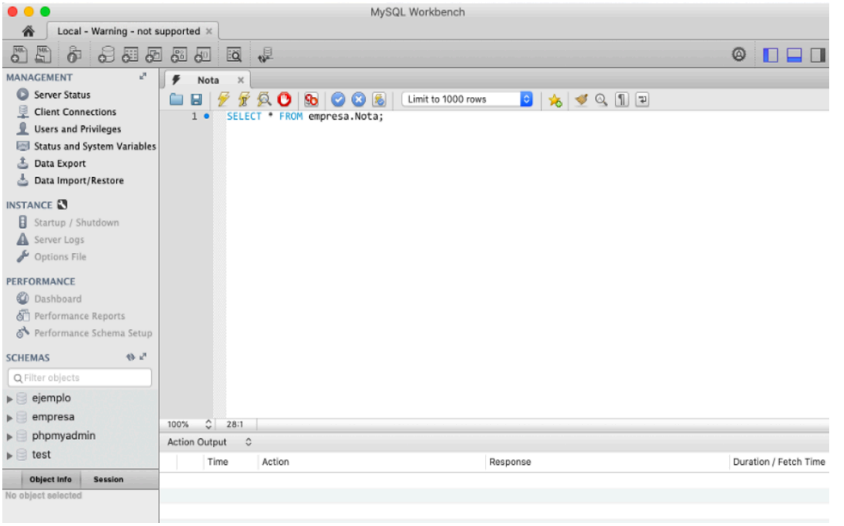


Ilustración 21. Creación de bases de datos

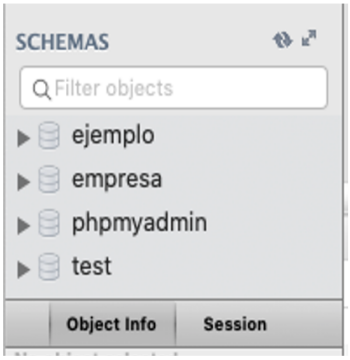


Ilustración 22. Existencia de objetos en esquemas

Tablas

Para comenzar el desarrollo de tabla primero se debe entrar en la base de datos a trabajar. Para ello se usa el siguiente código:

```
>  (query-exec pgc "create database empresa")
```

Luego de seleccionar la base de datos se debe determinar la forma de la tabla, los campos a usar y la información a tabular. Para promover el uso de la mayor parte de funciones, se creará una tabla *Nota* que almacenará la información de los estudiantes de acuerdo con un identificador numérico, un nombre, y una calificación de 1 a 10.

Para su creación se puede tomar en perspectiva la estructura de definición de funciones convencionales, de la siguiente manera:

```
>  (query-exec pgc "create table Nota
   (idAlumno integer primary key, nombre varchar(20),
   calificacion integer)")
```

Se crea una consulta guiada a la conexión *pgc*. Entre comillas se usan funciones SQL para crear una tabla de nombre *Nota* con los campos *idAlumno* que almacenara los identificadores numéricos para cada estudiante, nombre y calificación.

El campo *idAlumno* se define con *primary key*, es decir, llave primaria, que servirá para priorizar el uso de los datos almacenados en este campo al momento de ejecutar alguna consulta de selección, inserción, o eliminación. Por otro lado, *nombre* se crea con capacidad para 20 caracteres, similar a la definición de un vector, mientras que *calificación* se define como un entero.

Para comprobar la creación de la tabla se explora la pestaña *Schema* donde estarán todas las bases de datos creadas. Cada una almacenará la tabla, que a su vez podrá mostrar las filas, indexaciones, llaves, etc.

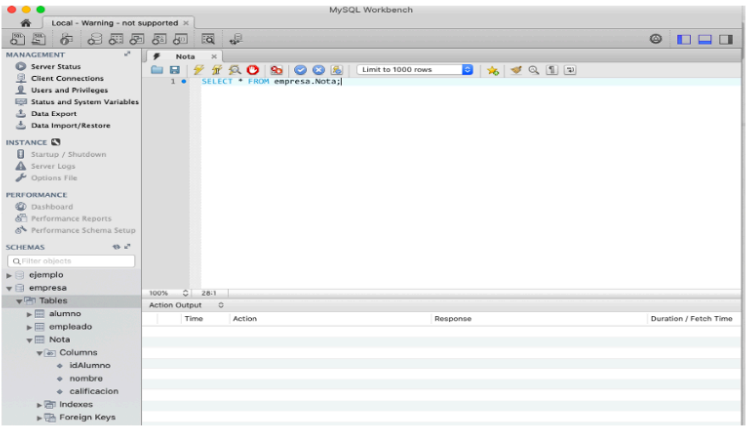


Ilustración 23. Creación de tablas

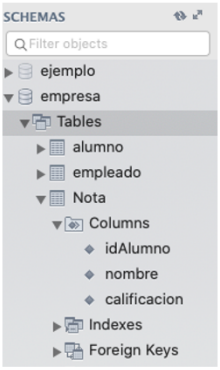


Ilustración 24. Existencia de objetos en esquemas

Nota: Si la lista no se actualiza automáticamente, presione las flechas que están junto al título Schema.

9.4 Desarrollo de funciones

Inserción de datos

Para la inserción de datos se usan las funciones *insert, into y* values, por ejemplo:

```
> (query-exec pgc "insert into Nota values (1, Jose, 5.6)")
```

Se desarrolla una consulta que inserta valores en la tabla Nota. Las partes como *into* y *values* funcionan como indicador o ruta de almacenamiento y como proceso para señalar los valores que se insertan en la tabla respectivamente.

La inserción de datos se hace en el orden de declaración de los mismos, siendo el identificador de alumnos de

primer lugar, nombre de segundo y calificación de tercero y último dato.

Ejercicio: Llene la tabla con un cierto número de datos aleatorios.

Visualización de la tabla

Se pueden visualizar los datos de las siguientes maneras:

- Usando la terminal de Dr Racket con el siguiente código:

```
>  (query pgc "select * from empresa.Nota")
```

Como resultado se obtiene:

```
>  (rows-result
    '(((catalog . "def")
       (database . "empresa")
       (table . "Nota")
       (original-table . "Nota")
       (name . "idAlumno")
       (original-name . "idAlumno")
       (length . 11)
       (typeid . long)
       (flags not-null primary-key))
      ((catalog . "def")
       (database . "empresa")
       (table . "Nota")
       (original-table . "Nota")
       (name . "nombre")
       (original-name . "nombre")
       (length . 60)
       (typeid . var-string)
       (flags))
      ((catalog . "def")
       (database . "empresa")
       (table . "Nota")
       (original-table . "Nota")
       (name . "calificacion")
       (original-name . "calificacion")
       (length . 12)
       (typeid . float)
       (flags)))
    '(#(1 "Miguel" 9.5)
      #(2 "jose" 10.0)
      #(3 "Maria" 5.400000095367432)
      #(4 "Victoria" 9.899999618530273)
      #(5 "Ester" 7.5)
      #(6 "Emanuel" 1.600000023841858)))
```

Nota: MySQL usa por defecto +15 dígitos de precisión para datos numéricos de tipo flotante.

•Visualizando por medio de MySQL workbench presionando el icono de tabla que se encuentra junto al título de la tabla (pestaña *Scheme*):

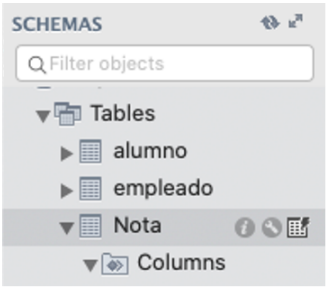


Ilustración 25. Visualización de tab

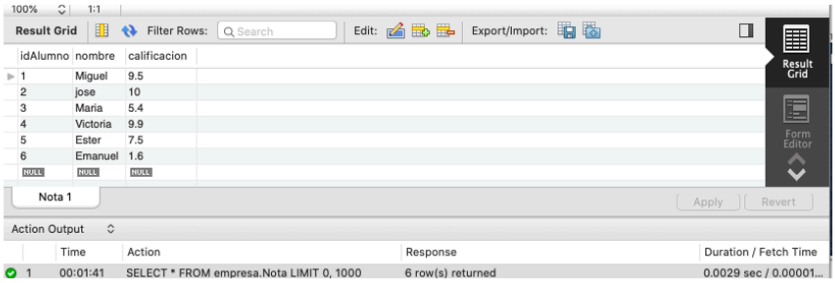


Ilustración 26. Tabla de MySQL Workbench

Se observan todos los campos con sus respectivos datos, incluso la siguiente casilla denotada por una etiqueta *null* indicando que se encuentra vacía.

En el campo *action output* se muestran todas las ejecuciones realizadas, y en este caso la selección de una tabla dentro de una base de datos, donde se indica la cantidad de filas utilizadas.

Limitando la visualización de los datos

Las operaciones de selección permiten priorizar los datos a visualizar. Para obtener toda la información

tabulada se usa un asterisco (*) indicando que se deben de tomar todos los datos contenidos en la tabla. Esta sección puede ser cambiada por el tipo de dato a tomar, por ejemplo, para obtener solamente los nombres de la tabla se usa:

```
>  (query pgc "select nombre from Nota")
```

Como resultado se obtiene:

```
>  (rows-result
    '(((catalog . "def")
       (database . "empresa")
       (table . "Nota")
       (original-table . "Nota")
       (name . "nombre")
       (original-name . "nombre")
       (length . 60)
       (typeid . var-string)
       (flags)))
    '(#("Miguel") #("jose") #("Maria") #("Victoria")
     #("Ester") #("Emanuel")))
```

La función *where* y los operadores de comparación (>, <, >=, <= y ¡= o <>) permiten filtrar detalladamente los datos a manejar. Como ejemplo, de la actual tabla se debe tomar los nombres con las calificaciones menores a 5, para ello se hace lo siguiente:

•Añadir nueva información para tener más datos que cumplan con la condición inicial, en este caso dos nuevos alumnos con notas menores a cinco.

•Utilizar la función de selección y condicionar la consulta con *where* y los operadores comparativos de la siguiente manera:

```
> (query pgc "select nombre,calificacion from Nota where
  calificacion < 5")
```

Se debe obtener:

```
> (rows-result …
   '(#("Emanuel"          1.600000023841858)          #("Sara"
  2.200000047683716) #("Joel" 3.4000000953674316)))
```

Se puede observar al final los nombres de los alumnos que tienen una calificación por debajo de 5. De esta forma se pueden variar las condiciones, ya sea para nombres, identificadores, o cualquier otro campo en uso.

9.5 Actualización y borrado de información

Actualización

Para la actualización de datos (específicamente la información de un registro) en tabla se usa la función *update*, para ello, se propone como ejercicio cambiar la nota de algún alumno ingresado usando lo siguiente:

```
>(query pgc "update Nota set calificacion = 7.8 where
 idAlumno = 6")
```

La modificación se hizo para el alumno con un identificador *idAlumno = 6*, donde su nota actual fue cambiada por un 7.8. Note que se usan otros elementos como *set* que asigna los nuevos valores, mientras que *where* especifica el registro que se debe cambiar, referenciándose a partir de otros campos distintivos como *idAlumno*.

Borrado

La función *delete* se encarga de borrar registros por completo, para esto se escogerá un registro aleatorio y se construirá lo siguiente:

```
>  (query pgc "delete from Nota where idAlumno = 4 ")
```

Como resultado se obtiene:

```
>   (rows-result …
    '(#(1 "Miguel" 9.5)
      #(2 "jose" 10.0)
      #(3 "Maria" 5.400000095367432)
      #(5 "Ester" 7.5)
      #(6 "Emanuel" 7.800000190734863)
      #(7 "Sara" 2.200000047683716)
      #(8 "Joel" 3.4000000953674316)))
```

Hay que ser cuidadoso al momento de usar *where* para determinar la información a borrar. En algunos casos puede suceder que se escoge un dato repetido múltiples veces y como resultado se tiene la eliminación de varios registros. Por defecto Workbench viene con opciones de modo seguro para estas prácticas, sin embargo, se debe de tener muy en cuenta las referencias utilizadas para la eliminación de datos para así evitar la pérdida de información.

9.6 Otras funciones y operaciones de SQL

Like

Esta función permite encontrar datos que cumplan con las coincidencias proporcionadas. *Like* (en ingles “como, parecido, de comparación”) utiliza información específica

como números o caracteres para comparar u retornar datos que contengan los elementos dados.

Como ejercicio, se debe de buscar en la base de datos todos los nombres que contengan la letra ‘e’. Para ello se hace:

```
> (query pgc "select nombre from Nota where nombre like
  'e%' ")
```

Se obtiene:

```
> (rows-result …
  '(#("Ester") #("Emanuel")))
```

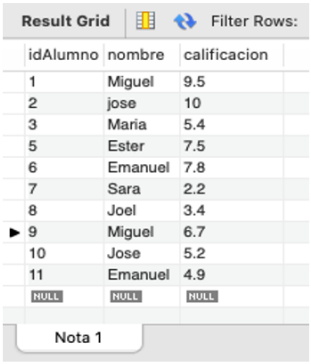
El signo de porcentaje utilizado luego de la letra ‘e’ representa el resto de información que se encuentre luego del dato o carácter. Pueden hacerse varios ejemplos y con distintas uniones entre datos como consultar los nombres que tengan la pareja ‘ue’, sin embargo, la función estrictamente determina si los datos evaluados contienen o no la condición pedida, por lo que no se examina desde una posición.

Ejercicio: Crear consultas que retornen datos con las siguientes solicitudes:

- Nombre de alumnos que contengan la unión ‘ia’
- Nombre de alumnos que tengan como letras finales ‘el’.

Distinct

El uso de *distinct* permite mostrar valores de un solo tipo, es decir, obviar los datos que estén repetidos. Por ejemplo, se tiene la siguiente tabla:



Result Grid | Filter Rows:

idAlumno	nombre	calificacion
1	Miguel	9.5
2	jose	10
3	Maria	5.4
5	Ester	7.5
6	Emanuel	7.8
7	Sara	2.2
8	Joel	3.4
▶ 9	Miguel	6.7
10	Jose	5.2
11	Emanuel	4.9
NULL	NULL	NULL

Nota 1

Ilustración 27. Tabla con información

Para poder visualizar todos los nombres sin repeticiones se usa:

```
>  (query pgc "select distinct nombre from Nota")
```

Se obtiene:

```
>  (rows-result …
   '(#("Miguel") #("jose") #("Maria") #("Ester") #("Emanuel")
   #("Sara") #("Joel")))
```

Ejercicio: *Usar la función selección para mostrar todas las notas sin repeticiones.*

Order by

Order by es utilizada cuando se requiere organizar datos de forma ascendente o descendente. Por ejemplo:

```
>  (query pgc "select * from Nota order by calificacion")
```

Se hace una consulta que ordenara las calificaciones de los alumnos almacenados. Por defecto se ordena ascendentemente, pero se puede especificar usando *ASC* (ascendente) o *DESC* (descendente).

Como resultado se obtiene:

```
>  (rows-result …
   '(#(7 "Sara" 2.200000047683716)
     #(8 "Joel" 3.4000000953674316)
     #(11 "Emanuel" 4.900000095367432)
     #(10 "Jose" 5.199999809265137)
     #(3 "Maria" 5.400000095367432)
     #(9 "Miguel" 6.699999809265137)
     #(5 "Ester" 7.5)
     #(6 "Emanuel" 7.800000190734863)
     #(1 "Miguel" 9.5)
     #(2 "jose" 10.0)))
```

De igual forma que se ordenan los datos numéricos, sucede con los caracteres, donde se posicionan de acuerdo con el orden alfabético conocido, por ejemplo:

```
>  (query pgc "select * from Nota order by nombre desc")
```

Como resultado:

```
>  (rows-result …
   '(#(7 "Sara" 2.200000047683716)
     #(9 "Miguel" 6.699999809265137)
     #(1 "Miguel" 9.5)
     #(3 "Maria" 5.400000095367432)
     #(2 "jose" 10.0)
     #(10 "Jose" 5.199999809265137)
     #(8 "Joel" 3.4000000953674316)
     #(5 "Ester" 7.5)
     #(6 "Emanuel" 7.800000190734863)
     #(11 "Emanuel" 4.900000095367432)))
```

9.7 Operaciones entre tablas

Truncate table

Se tiene la siguiente tabla:

```
>(rows-result …
  '(#(1 "Esteban" "Mesero")
    #(2 "Jose" "Gerente")
    #(3 "Ester" "Subgerente")
    #(4 "Maria" "Mecanico")
    #(5 "Pedro" "Bombero")
    #(6 "Sofia" "Medico")
    #(7 "Miguel" "Programador")
    #(9 "Luis" "Auxiliar contable")))
```

Truncate table se encarga de limpiar todos los registros de una tabla. Por ejemplo:

```
>(query pgc "truncate table empleado")
```

Como resultado se obtiene:

```
> (query pgc "select * from empleado")

> (rows-result …
  '())
```

Funciones propias de Dr Racket

Algunas de las funciones más usadas son:

```
1   (query-row pgc "select nombre from Nota where nota = 7.5")
2
3   (query-list pgc "select distinct nombre from Nota where
    nombre = 'Jose'")
4
5   (query-value pgc "select nombre from Nota where idAlumno =
    8")
6
7   (list-tables pgc #:schema 'search-or-current)
8
9   (table-exists? pgc "empleado")
10
11  (sql-null? (query-value pgc "select null"))
```

Como resultado se obtiene:

```
>  '#("Ester")
>  '("jose")
>  "Joel"
>  '("Nota" "alumno" "empleado")
>  #t
>  #t
```

Cabe resaltar que la mayoría de estas funciones son complementarias al funcionamiento general de consultas, colaborar en la reducción de retorno de información mostrando solo el resultado necesario, entre otras. Del mismo modo pueden colaborar en la fabricación de un programa autosuficiente que maneje los datos de algún servicio en específico como páginas web, aplicaciones empresariales, o simplemente gestores de data.

9.8 Problemas de seguridad de la información

Inyección SQL

En seguridad informática se conoce como un método de infiltración o inyección de código de consulta a una aplicación o servicio web con el fin de modificar los datos o procedimientos del objeto víctima.

La vulnerabilidad es causada por la mala verificación de entrada de variables en un programa, es decir, interpretar información literal (datos definidos de los formularios) en instrucciones de funcionamiento, lo que brinda la posibilidad de generar código ajeno en tiempo de ejecución, arriesgando la seguridad de la información almacenada.

Funcionamiento

Mediante el ingreso de datos como los formularios, un atacante puede implementar consultas añadiéndolas en el campo de información, por ejemplo, una compañía almacena las claves y usuarios de banco en una base de datos conectada a través de una página web; una persona decide registrarse para obtener una cuenta por lo que decide rellenar la información pedida por un formulario de entrada. En el campo usuario la persona decide ingresar "user '; DROP TABLE usuarios'". Al verificar la información validada el servicio de almacenamiento cataloga la información dada como una instrucción oficial para las bases de datos, y en este caso la de eliminar la tabla que almacena cada usuario registrado.

Tipos de inyección SQL

- **Orden de inyección**

Se dividen en dos grupos:

- **Primer orden:** Se obtienen resultados instantáneos luego de la inyección del código malicioso.
- **Segundo orden:** Su objetivo es dejar rastros de código intruso para ser usado luego de la intrusión.

- **Canal de extracción de datos:**

Se tienen las siguientes variantes:

- **In-band:** Se utiliza el mismo canal de comunicación para atacar (introducir código) como recoger resultados.
- **Out-of-band:** Se utilizan canales distintos tanto para el ataque como para la obtención de resultados.

- **Respuesta del servidor:**

Se tienen las siguientes variantes:

- **Error-based:** Muchas de las operaciones SQL, en concordancia con el servidor proveen información descriptiva del manejo de los datos como el nombre de las tablas. Este tipo de intrusión se basa en provocar errores para conocer la estructura de la base de datos, de esta

manera se conoce a profundidad el ambiente para poder así realizar ataques más complejos.

•**UNION query-based:** a través de un operador *UNION*, se concatena información usando la adición entre consultas, es decir, mediante una consulta original insertada, se le añade nuevas peticiones, las cuales darán como resultado la unión de información de solicitudes distintas. Como ejemplo se tiene la siguiente consulta: "*select nombres from users_table where id < '12' unión select username, password from users_table*".

Aquella consulta concatena dos solicitudes, la primera encargada de seleccionar los nombres de una tabla *users_table* donde su id sea menor a 12, y la segunda se encarga de seleccionar el *username* y *password* de la tabla *users_table*.

•**Time Based Blind Injection:** Igual que el método anterior pero se usan procesos de automatización de consultas para evaluar el comportamiento de la base de datos y él envió de respuestas a la página web o servicio que la ejecute en función del tiempo.

•**Boolean Base Blind injection:** Se extraen datos mediante secuencias booleanos verdaderas y falsas. Se evalúan las vulnerabilidades de las entradas del usuario mediante las siguientes sentencias:

- ... and 1 = 1.
- ... and 1 = 2.

Protección contra ataques de inyección SQL

Algunos métodos de seguridad contra intrusos son:

- Restringir el uso o exposición de la base de datos en el servidor, es decir, eliminar el acceso a todas las direcciones que maneje el servidor en el cual se conecta la base de datos.
- Utilizar usuarios específicos, con modelos de cifrados, o palabras muy específicas.
- Bloquear el uso de ciertos caracteres al momento de ingresar datos.
- Utilizar consultas preparadas, es decir, la ejecución de una misma sentencia SQL. Estas reducen el tiempo de análisis de la petición (similares al uso de funciones, donde son creadas una sola vez, pero ejecutadas múltiples ocasiones) y los parámetros usados minimizan el ancho de banda usado por el servidor, evitando que la información se escape a otros lugares de dirección del servidor.
- Prohibir la concatenación de texto.

Nota: Para las consultas preparadas y el uso de otras funciones en este ámbito, visitar la documentación oficial de Dr Racket en bases de datos.

A continuación, se facilita una serie de trucos para practicar una inyección SQL:

Propósito	Ejemplo
Valor ASCII-> carácter	`select char(65); #returns A`
Carácter -> valor ASCII	`select ascii('A'); #returns 65`
Casting	`select cast('123' AS char);`
Concatenación entre cadenas	`select concat('A', 'B'); #returns AB`
Estructura if	`select if(1 = 1, 'foo', 'bar'); returns 'foo'`
Evitando comillas	`select 0x414243; #returns ABC`
Retrasar el tiempo	`select sleep (5)`
Acceso de archivos locales	`select * from tabla into dumpfile'/ruta de archivo'; escribe la información en el Sistema del atacante`
Hostname, dirección IP	`select @@hostname;`
Acceso	`grant all privileges on *.* to 'user';`
Crear usuarios	`create user usuario;`
Borrar usuarios	`drop user usuario;`

Tabla 7. Sintaxis para inyección SQL.

10. Seguridad informatica

10.1 Criptografía

Rama de la criptología que estudia las técnicas de cifrado guiadas a la alteración y obtención de representaciones lingüísticas (textos, imágenes, etc.) haciéndolos irreconocibles a receptores no autorizados. El objetivo de la criptografía es proporcionar seguridad, confidencialidad e integridad en la información, usando algoritmos y estructuras sistemáticas que permitan mantener el anonimato de un conjunto de información.

Tipos de encriptación

Dependiendo del modelo que se use para modificar un mensaje existen tres tipos:

- **Simétrica:** También llamada criptografía de una sola clave o de clave privada. Para el cifrado y descifrado de información se utiliza una misma clave, la cual se comparte solo entre el emisor y el receptor.
- **Asimétrica:** También llamada criptografía de clave pública es un método de seguridad que utiliza una clave pública y una privada para el receptor y emisor. Al cifrar un mensaje, el receptor utiliza la clave pública del emisor para el envío, y este último es el único que podrá visualizar la información con la clave privada que le es otorgada. Esta se crea como solución al problema de intercambio de claves y seguridad con el uso de una sola clave para dos destinos.

En la siguiente imagen se puede apreciar el funcionamiento de los dos modelos:

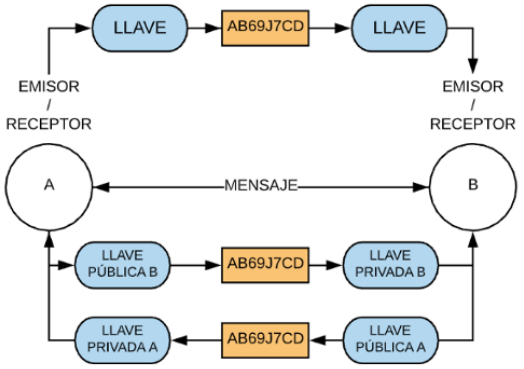


Ilustración 28. Sistema simétrico y asimétrico de encriptación

•**Hibrida:** Utiliza conceptos de la criptografía simétrica y asimétrica. La rapidez de cifrado para la primera es complementaria con la seguridad de la información que brinda la segunda.

10.2 Operabilidad

Ejecución

Para disponer de toda la sintaxis, primero se debe instalar el paquete de funciones *crypto*, para ello hay que hacer lo siguiente:

•Dirigirse a la pestaña *File*
•Buscar la opción *install package*

- Buscar el nombre *crypto*.
- Instalar el paquete.

Nota: Estos pasos son igualmente útiles para los capítulos donde se utilizan nuevos paquetes de funciones.

Luego de haber instalado la librería se utiliza *(require crytpo)* Para carga las funciones en el entorno de Dr Racket.

La librería de criptografía maneja las operaciones generales, sin embargo, para mayor cobertura de acciones, se instalará de la misma manera que con *crypto* una nueva librería subyacente, *crypto/libcrypto*.

Al final se debe tener:

```
1  #lang Dr Racket
2
3  (require crypto)
4  (require crypto/libcrypto)
```

10.3 Resúmenes de mensajes

Llamadas también hashes criptográficos, son un algoritmo matemático que transforma cualquier bloque de información en una nueva serie de caracteres (independiente de la longitud de información a transformar, el tamaño de la serie puede variar). Su uso se confiere a la seguridad de contraseñas y usuarios, el ocultamiento y la integridad de la información tratada, detección de programa maligno, entre otras.

Para la implementación de este método Dr Racket contiene las siguientes funciones:

Función	Ejemplo	Descripción
digest-spec?	(digest-spec? v)	Función booleana que evalúa si v es un resumen de mensaje. Estos se nombran como ‘blake2b-160, ‘blake2b-256, ‘sha1, entre otros.
get-digest	(get-digest <digest-spec> <proveedor de procedimientos>)	Implementa la función hash criptográfica usando métodos definidos por el proveedor.
digest-size	(digest-size <digest-spec>)	Retorna en bytes el tamaño del mensaje resumido.
	FUNCIONES DE ALTO NIVEL	
digest	(digest <digest-spec> <input> <#:key key>)	Calcula un nuevo mensaje con la función hash <digest-spec> del mensaje introducido. La llave es opcional, si se es posible usarla con otro hash (Blake, entre otras)

Tabla 8. Funciones para hash criptográficos.

NOTA: las funciones hash (es decir, digest-spec) vienen prediseñadas en distintos grupos. Dr Racket provee los siguientes:

- `'blake2b-160`
- `'blake2b-256`
- `'blake2b-384`
- `'blake2b-512`
- `'blake2s-128`

- 'blake2s-160
- 'blake2s-224
- 'blake2s-256
- 'md2
- 'md4
- 'md5
- 'ripemd160
- 'sha0
- 'sha1
- 'sha224
- 'sha256
- 'sha3-224

Ejercicio No. 1

```
1  #lang Dr Racket
2
3  (require crypto)
4  (require crypto/libcrypto)
5
6  (define encriptar (get-digest 'sha1 libcrypto-factory))
7
8  (digest encriptar "Hello world!")
```

En la linea 6 se crea *encriptar* que se compone de usar la función de resumen, un hash propiciado por la librería *crypto* y el conjunto de algoritmos de hash dado por *libcrypto-factory*. Al final se usa *digest* para juntar las definiciones anteriores y encriptar el mensaje *“Hello world!”*.

El resultado final es:

```
>  #"\323Hj\351\23nxV\274B!#\205\352yp\224GX\2"
```

10.4 Cifrado Simétrico

Se usa una clave secreta para cifrar un texto de entrada en una cadena de caracteres que puede ser de igual longitud o aproximada. Los algoritmos usados se clasifican en familias (particulares cada una en la cantidad de bytes usados) como DES, RC4, DEA, AES, CTR, etc.

Nota: Se recomienda consultar las familias existentes y que modos componen a cada una.

Algunas de las operaciones esenciales que provee el ambiente de Dr Racket son:

Función	Ejemplo	Descripción
`cipher-spec?`	(cipher-spec? v)	Función booleana que evalúa si v es un elemento de cifrado. Para determinar si es una función de cifrado, debe mantener la siguiente estructura: ●Para control de flujo: (<'método de flujo o familias> <'stream>) ●Para bloques: (<'metodo de bloqueo familias> <'modo>)

Función	Ejemplo	Descripción
get-cipher	(get-cipher <cipher-spec> <proveedor de procedimientos >)	Determina si el cifrado tomado hace parte del conjunto de operaciones brindado por el proveedor de métodos.
cipher-block-size	(cipher-block-size <cipher-spec>)	Retorna en bytes el tamaño del texto manipulado.
cipher-default-key-size	(cipher-default-key-size <cipher-spec>)	Retorna el tamaño en bytes de la clave usada para el cifrado
cipher-key-sizes	(cipher-key-sizes <cipher-spec>)	Retorna los posibles tamaños de las claves usadas.
cipher-aead?	(cipher-aead? <ciper-spec>)	Función booleana que verifica si el cifrado es una encriptación autentica, es decir, seguridad y garantías sobre la información oculta.
generate-cipher-key	(generate-cipher-key <cipher-spec> <#:size size>)	Genera un clave aleatorio para la función de cifrado

Función	Ejemplo	Descripción
generate-cipher-iv	(generate-cipher-iv <cipher-spec>)	Genera un vector de inicialización.
FUNCIONES DE ALTO NIVEL		
encrypt	(encrypt <cipher-spec> <llave> <vector> <input> <#: add, #:auth-size, #:pad>)	Al igual que las declaraciones anteriores, estas funciones forman en conjunto el cifrado simetrico de un texto o información introducida.
decrypt	(dencrypt <cipher-spec> <llave> <vector> <input> <#: add, #:auth-size, #:pad>)	

Tabla 9. Funciones para cifrado simétrico

Notas:

•*El cifrado simétrico maneja la seguridad en dos entornos, flujo de información, que se modifica la representación bit a bit, y por bloques, lo que significa que el texto a ocultar es dividido en bloques de igual tamaño, y sobre los cuales se usa la clave otorgada. El tamaño de cada bloque es indicado por un vector de inicialización, un bloque de bits.*

•*Para el cifrado por flujo se tiene los siguientes métodos (familias):*

- 'chacha20
- 'chacha20-poly1305
- 'chacha20-poly1305/iv8

- 'rc4
- 'salsa20
- 'salsa20r12
- 'salsa20r8
- 'xchacha20-poly1305.

Para el cifrado por bloques se tienen los siguientes métodos (familias):

- 'aes
- 'blowfish
- 'camellia
- 'cast128
- 'des
- 'des-ede2
- 'des-ede3
- 'idea
- 'serpent
- 'twofish,

Ejemplo No. 1

```
1   #lang Dr Racket
2
3   (require crypto crypto/all)
4   (use-all-factories!)
5
6
7   (cipher-spec? '(aes gcm))
8
9   (define llave (generate-cipher-key '(aes gcm)))
10
11  (define vector (generate-cipher-iv '(aes gcm)))
12
13  (define encriptar (encrypt '(aes gcm) llave vector "Hola
    mundo!"))
14
15  encriptar
16
17  (decrypt '(aes gcm) llave vector encriptar)
```

En este ejercicio se pueden utilizar todos los proveedores de métodos que proporciona la librería *crypto*. *¡Use-all-factories!* se encarga de proporcionar al entorno todas las operaciones con el fin de generar la mayor cantidad de procedimientos posibles para llevar a cabo un buen cifrado.

Se quiere utilizar como método de cifrado el *‘(aes gcm)* (en otras palaras, el tipo de algoritmo es Advanced Encryption Standard con un modo en bloques Galois/ Counter Mode), y con *cipher-spec?* se evalúa si esta forma es un método de cifrado valido. Se crean las funciones de llave y vector las cuales son complementarias al uso de la función principal de *encriptar*. Al final se llama esta última función, que presenta el mensaje entre caracteres literarios y números enteros, al igual que lo hace *decrypt* que devuelve al estado inicial la información cifrada.

El resultado debe ser el siguiente:

```
> #t
> #"Y\261(\2040\363\324K\342{\325/\240\24\234\3369p\252
  ~V\211}\317\345=\363”
> ”#"Hola mundo!"
```

Matemáticas y algoritmos

Se explicará la librería de matemáticas, con temas como números flotantes, funciones complejas, matrices, entre otros. De igual manera se detallará la relación entre dichas temáticas con los algoritmos, su concepto, características y resolución.

11. Algoritmos y álgebra

11.1 Concepto de algoritmo

En matemáticas, ciencias de la computación, lógica y relacionadas, un algoritmo se conoce como un conjunto de instrucciones no-ambiguas, ordenadas y finitas que permiten la solución de un problema o la realización de cierta actividad. En programación los algoritmos son una secuencia de pasos lógicos que permiten desarrollar un programa.

Son fundamentales en la vida cotidiana para el desarrollo de ciertas habilidades como las matemáticas con las tablas de multiplicar, operaciones de cálculo, o el mismo algebra con sus operaciones aritméticas y matriciales. No solo en ciencias exactas, también en actividades particulares como un manual de instrucciones, una receta de cocina o atarse los zapatos.

Características

Algunas propiedades que caracterizan a un proceso como algoritmo son:

- Debe ser preciso y ordenado, avanzando en un tiempo secuencial, paso a paso.
- Debe ser definido, independiente del número de veces que se realice, el resultado será siempre el mismo

- Debe ser finito
- Debe tomar una cantidad de datos límite entre paso y paso

Clasificación

Los algoritmos se dividen de la siguiente forma:

- Según su sistema de **signos**, que usan caracteres verbales, matemáticos y computacionales. Algunos modelos de este tipo son:

- **Cualitativos:** Instrucciones que se describen usando palabras. Como ejemplo existen las recetas de cocina o las manuales de instrucciones.
- **Cuantitativos:** Se usan números y operaciones matemáticas para encontrar un resultado o configurar un procedimiento.
- **Computacionales:** Instrucciones de alta complejidad ejecutados por una computadora. Generan un algoritmo cuantitativo optimizado.
- **No computacionales:** Algoritmos manuales, no necesitan ayuda computacional.
- Según su objetivo, ya sea optimización, dinamismo, incremento, búsqueda, entre otros. Usados comúnmente en ámbitos empresariales y comerciales. Se tienen los siguientes modelos:

- **Marcaje:** Estudia al cliente, evalúa sus actitudes frente a distintas circunstancias (precios de lanzamiento, rebajas, aumentos, etc.) y crea un sistema que genera las posibles opciones de venta que más se acomoden al consumidor.

•**Programación dinámica:** Resuelve situaciones de gran magnitud, dividiéndolo en varios factores hasta que se encuéntrala una solución.

•**Vuelta atrás:** Usa la técnica y el análisis para operaciones de mercado, precios, tráfico de personas, entre otras, y analiza el impacto de estos con el tiempo. El mercado de valores maneja múltiples indicadores que evalúan el impacto del comportamiento de las acciones.

•**Ordenamiento:** Organizan una secuencia de datos numéricos en un orden especifico.

•**Búsqueda:** Encuentra dato especifico en un conjunto de información.

•**Encantamiento:** Ayuda a transmitir un mensaje al público de maneras específicas. Muy usado para técnicas de marketing y publicidad.

•Según la estrategia usada para el alcance de un resultado son:

•**Probabilístico:** Estima la probabilidad cierta o incierta de algún proceso o situación particular.

•**Cotidianos:** Describen las actividades usuales de una persona.

•**Escalada:** A través de varios resultados, por ensayo y error va progresando hasta encontrar un resultado satisfactorio.

Algunos de los algoritmos más usados en programación son:

•Ordenamiento de burbuja.

```
Procedimiento OrdenarBurbuja (a0,a1,a2,..., an)
    Para i = 0 hasta  (n-1) hacer
        Para j = i +1 hasta n hacer
            Si (an >, < an+1) entonces
                Aux = an
                an = an+1
                an+1 = Aux
            fin Si
        fin Para
    fin Para
fin Procedimiento
```

•Algoritmo de Dijkstra (grafos, camino más corto).

1.Seleccionar el vértice de partida (origen).

2.Marcar el punto de partida como el punto de inicio.

3.Determinar los caminos especiales desde el nodo de partida. Un camino especial es aquel que se traza a través de nodos ya marcados.

4.Para cada nodo no marcado se debe escoger si usar un camino especial o calcular uno nuevo.

5.Para seleccionar un nuevo nodo no marcado como referencia, se debe tomar aquel cuyo camino especial para encontrarlo es el mínimo.

6.Cada camino corresponde a la suma de los pesos de las aristas que forman el camino para llegar del nodo principal al resto de nodos, pasando únicamente por caminos especiales.

11.2 Algoritmos con Dr Racket

Los procedimientos que se mostraran hacen parte de las librerías *algorithms* y *math*. Muchos de los procesos son encapsulados en una sola función o estructura por lo que se recomienda manejar las pruebas de escritorio (análisis a mano) para conocer más a fondo su funcionamiento.

Nota: (require algorithms) para cargar la librería de algoritmo. Para cargar la librería de matemáticas se tienen las siguientes:

- Math/base para constantes y funciones elementales
- Math/flonum para funciones flotantes
- Math/special-functions para funciones especiales
- Math/number-theory para funciones de teoría de números
- Math/matrix para operaciones matriciales en algebra lineal

Nombre	Función	Descripción
	Algoritmos	
`adjacent-map`	`(adjacent-map <lista> <proceidmiento>)`	Aplica un procedimiento a cada elemento de una lista.
`chunks-of`	`(chunks-of <lista> <n>)`	Retorna una lista de listas de n elementos cada una.
`generate`	`(generate <n> <procedimiento>)`	Retorna una lista de n números después de realizar el procedimiento n veces.

Nombre	Función	Descripción
increasing?	(increasing? <lista>)	Función booleana que determina si los elementos de una lista están en orden creciente.
repeat	(repeat <n> <valor>)	Crea una lita con valores repetidos n veces.
sliding	(sliding <lista> <tamaño> <paso>)	Retorna una lista de listas, con un tamaño indicado cada una y una separación de elementos dada.
sorted?	(sorted? <ista>)	Función booleana que evalúa si una lista esta ordenada o no.
zip	(zip <lista> <lista>)	Retorna una lista con sublistas que contienen los pares según la posición de cada elemento en cada lista.

Matemáticas		
	math / base	
Euler Proporción aúrea Gamma Catalan	euler.√ phi.√ gamma.√ catalan.√	Algunas de las constantes más conocidas y usadas en matemáticas aplicadas. Alojan cantidades más precisas al valor que comúnmente les corresponden.
float-complex?	(float-complex? <v>)	Evalúa si es un flotante complejo

Matemáticas		
number->float-complex	(number->float-complex <v>)	Convierte cualquier número en un flotante complejo
absolute-error	(absolute-error <número> <número>)	Retorna el error o diferencia absoluta entre dos números
relative-error	(relative-error <nuúmero> <número>)	Retorna el error relativo (más aproximado) entre dos cantidades.
math / flonum		
fleven? flodd?	(fleven? <número flotante>) (flodd? <número flotante>)	Evalúan si los números flotantes ingresados son pares o impares.
flrational?	(flrational? <número flotante>)	Evalúa si el número ingresado contiene raíz cuadrada.
flinfinite?	(flinfinite? <número flotante>)	Evalúa si el número flotante es infinito.
math / special-functions		
gamma	(gamma <número>)	Calcula la funión gamma, generalización de la función factorial.
psi.√	(psi.√ <número>)	Calcúla la función digamma, derivada logarítmica de la función gamma

math / special-functions		
psi	(psi <número entero> <número real>)	Calcúla la función poligámica, derivada logarítmica número m de la función gamma.
erf erfc	(erf <número real>) (erfc <número reeal>)	Calcúlan la función error y error complementario.
lambert	(Lambert <número real>)	Calcúla la función Lambert
zeta	(seta <número real>)	Calcúla la función zeta de Riemann
eta	(eta <número real>)	Calcúla la función de Dirichlet eta.
beta	(beta <número real><número real>)	Calcúla la función beta.
Fresnel-s Fresnel-c Fresnel-RS Fresnel-RC	(Fresnel-s <número real>) (Fresnel-c <número real>) (Fresnel-RS <número real>) (Fresnel-RC <número real>)	Calcúla las integrales de Fresnel.

math / number-theory		
`solve-chinese`	`(solve-chinese <lista de eneteros> <lista de eneteros>)`	Retorna el menor número natural que es una solución a las ecuaciones: • $x = a_1 \pmod{n_1}$ • $x = a_k \pmod{x_k}$
`divisors`	`(divisors <número entero>)`	Retorna todos los divisores de un número entero dado.
`fibonacci`	`(Fibonacci <número entero>)`	Retorna los Fibonacci con un rango de un número dado
`quadratic-solutions`	`(quadratic-solutions <número real><número real><número real>)`	Retorna una lista de todas las soluciones reales para la ecuación $ax^2 + bx + c$.

math / matrix		
`matrix`	`(matrix [[<datos>] [<datos>]])`	Crea una matriz.
`identity-matrix`	`(identity-matrix <tamaño nxn> <valores de la diagonal> <valores fuera de la diagonal>)`	Crea una matriz identidad

math / matrix		
matrix? row-matrix? col-matrix? square-matrix?	(matrix? <array>) (row-matrix? <array>) (col-matrix <array>) (square-matrix? <array>)	Retorna verdadero si el argumento es una matriz, matriz-cuadrada, matriz-fila o matriz-columna.
matriz-num-rows matriz-num-cols	(matriz-num-rows <matriz>) (matriz-num-cols <matriz>)	Retorna el número de filas y columnas respectivamente.
matrix-ref matrix-row matrix-col	(matrix-ref <matriz> <fila> <columna>) (matrix-row <matriz> <fila>) (matrix-col <matriz> <columna>)	Funciones que retornan el elemento en una posición dada, además de filas y columnas independientes.
diagonal-matrix	(diagonal-matrix <lista> <números fuera de la diagonal>)	Crea una matriz diagonal.
matrix-diagonal	(matrix-diagonal <matriz>)	Retorna un arreglo con los datos que componene la diagonal de la matriz dada.

math / matrix		
matrix-upper-triangle matrix-lower-triangle	(matrix-upper-triangle <matriz>) (matrix-lower-triangle <matriz>)	Retornan los triangulos superiores e inferiores de una matriz
list->matrix matrix->list	(list->matrix <filas> <columnas> <lista>) (matrix->list <lista>)	Vuelve una lista en una matriz y viceversa.
vector->matrix matrix->vector	(vector->matrix <filas> <columnas> <vector>) (matrix->vector <vector>)	Vuelve un vector en matriz y viceversa.
matrix+ matrix- matrix* matrix-scale	(matrix+ <matriz> …) (matrix- <matriz> …) (matrix* <matriz> …) (matrix-scale <matriz> <número>)	Operaciones bascias con matrices como suma, resta, multiplicación, multiplicación por escalar lambda.
matrix-map	(matrix-map <procedimiento> <matriz> <matriz> …)	Itera sobre cada posición o elemento en una matriz. Pueden ser una o varias matrices.

math / matrix		
matrix=	(matrix= <matriz> <matriz> …)	Retorna verdadero si las matrices propuestas son iguales en forma
matrix-transpose	(matrix-transpose <matriz>)	Retorna la transpuesta de una matriz
matrix-solve	(matrix-solve <matriz> <matriz>)	Retorna una matrix con los resultados de un sistema de ecuaciones para AX = B
matrix-invertible	(matrix-invertible? <matriz>)	Función booleana que determina si una matriz es invertible.
matrix-inverse	(matrix-inverse <matriz>)	Retorna la matriz inversa de una matriz, si existe.
matrix-determinant	(matrix-determinant)	retorna el determinante de una matriz cuadrada.
matrix-gauss-elim	(matrix-gauss-elim <matriz><jordan ?><hacer pivote 1><pivote a usar>	Aplica los algoritmos de eliminación Gaussiana y Gauss-Jordan para la resolución de matrices. -Sus valores predeterminados son #f.

math / matrix		
matrix-gram-schmidt	(matrix-gram-schmidt <matriz> <normalizada?>)	Retorna una matriz modificada por el algoritmo de gran shcmidt

Tabla 10. Funciones del conjunto de la librería math.

11.3 Ejercicios resueltos

Aplicación de algunas de las funciones anteriores:

Algoritmos

```
1   #lang Dr Racket
2
3   (require algorithms)
4
5   (chunks-of (list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15) 5)
6
7   (increasing? (list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15))
8
9   (repeat 6 34)
10
11  (sliding (list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15) 5 1)
```

Como resultado se obtiene:

```
> '((1 2 3 4 5) (6 7 8 9 10) (11 12 13 14 15))
  #t
  '(34 34 34 34 34 34)
  '((1 2 3 4 5)
    (2 3 4 5 6)
    (3 4 5 6 7)
    (4 5 6 7 8)
    (5 6 7 8 9)
    (6 7 8 9 10)
    (7 8 9 10 11)
    (8 9 10 11 12)
    (9 10 11 12 13)
    (10 11 12 13 14)
    (11 12 13 14 15))
```

Math/base

```
1   #lang Dr Racket
2
3   (require math/base)
4
5   euler.0
6   phi.0
7   gamma.0
8   catalan.0
9
10  (define num (number->float-complex 7))
11  (float-complex? num)
12  num
13
14  (absolute-error euler.0 phi.0)
15  (relative-error euler.0 phi.0 )
```

Como resultado se obtiene:

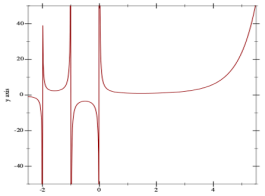
```
>   2.718281828459045
    1.618033988749895
    0.5772156649015329
    0.915965594177219
    #t
    7.0+0.0i
    1.1002478397091502
    0.679990560988901
```

Math/special-functions

Para una mejor representación de cada una de las funciones, se recomienda usar librerías de gráficos integrados como *plot*.

Nota: Se podrá ver la estructuración del modo gráfico, sin embargo, se recomienda visitar la documentación oficial para conocer más a fondo el funcionamiento de la librería.

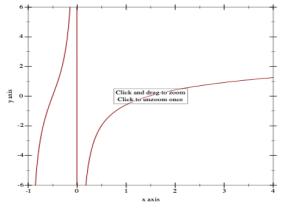
Función gamma



```
1 #lang Dr Racket
2 (require plot)
3 (require math/special-functions)
4 (plot (function gamma 2.5 5.5) #:y-
  min
5 50 #:y- max 50)
```

Ilustración 29. Ejemplo gráfico de la función gamma

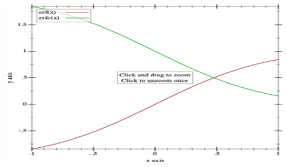
Función Psi0




```
1 #lang Dr Racket
2 (require plot)
3 (require math/special-functions)
4 (plot (function psi0 -1 4) #:y-min
  5-6
5 #:y-max 6
```

Ilustración 30. Ejemplo gráfico de la función Psi

Funciónes Erf y Erfc.




```
1  #lang Dr Racket
2  (require plot)
3  (require math/special-functions)
4  (plot (list (function erf -1 1
   #:label "erf(x)
5  (function erfc #:color 2 #:label
6     "erfc(x)")))
```

Ilustración 31. Ejemplo gráfico de las funciones Erf y Erfc

Math/number-theory

•Solución china

```
1  #lang Dr Racket
2
3  (require math/number-theory)
4
5  ; Cuál es el número x que dividido entre tres deja un
   residuo
6  ; de dos, cuando divide entre cinco deja un resto de tres,
7  ; y cuando divide entre siete da como resultado dos
8
9  (solve-chinese (list 2 3 2) (list 3 5 7))
```

Como resultado se obtiene:

```
>  23
```

•Ecuaciones de segundo grado

```
1   #lang Dr Racket
2
3   (require math/number-theory)
4
5   ;Encuentre las soluciones para las siguientes expresiones:
6
7   ;2x^2 + 3x + 6 = 0
8   (quadratic-solutions 2 3 6)
9
10  ;4x^2 + 18x + 9 = 0
11  (quadratic-solutions 4 18 9)
12
13  ;x^2 + 8x + 12 = 0
14  (quadratic-solutions 1 8 12)
```

Como resultado se obtiene:

```
>  '()
   '(-3.9270509831248424 -0.5729490168751576)
   '(-6 -2)
```

Math/matrix

Aplicación de funciones elementales para el uso del álgebra lineal:

```
1   #lang Dr Racket
2
3   (require math/matrix)
4
5   (define M1 (matrix [[1 2 3][5 6 7][0 8 3]]))
6
7   (matrix? M1)
8
9   (matrix-num-rows M1)
10  (matrix-num-cols M1)
11
12  (matrix-ref M1 1 2)
13
14  (matrix-diagonal M1)
15
16  (matrix-upper-triangle M1)
17  (matrix-lower-triangle M1)
18
19  (matrix-map sqr M1)
```

Como resultado se obtiene:

```
>  #t
   3
   3
   7
   #<array '#(3) #[1 6 3]>
   #<array '#(3 3) #[1 2 3 0 6 7 0 0 3]>
   #<array '#(3 3) #[1 0 0 5 6 0 0 8 3]>
   #<array '#(3 3) #[1 4 9 25 36 49 0 64 9]>
```

•Operaciones matriciales

Dr Racket proporciona muchas de las operaciones que se realizan en estas, como las siguientes:

```
#lang Dr Racket

(require math/matrix)

(define M1 (matrix [[34 9 8][ 9 3 6][ 87 2 6]]))
(define M2 (matrix [[2 65 3][ 2 9 2][ 56 76 32]]))

;Operaciones básicas
(matrix+ M1 M2)
(matrix- M1 M2)
(matrix* M1 M2)
(matrix-scale M1 4)

(matrix-transpose M2)

(matrix-solve M1 M2)

(matrix-gauss-elim M1)
```

Las primeras líneas hacen referencia a la suma, resta y multiplicación de matrices. Los pasos para proceder con cada una son:

- **Suma-resta**

Las matrices para operar deben de tener la misma dimensión. Se sumarán las posiciones iguales entre las matrices hasta abarcar todos los elementos.

- **Multiplicación**

Desde la definición teórica:

Si $(A = (a_{ij}) \in M_{m \times n})$ y $(B = (b_{st}) \in M_{n \times l})$, el producto se define como:

$AB = C \rightarrow (c_{it}) \in M_{m \times l}$, donde $c_{it} = f_i(A).c_t(B)$

Y su algoritmo:

- Comprobar que el número de columnas de una matriz A es igual al número de filas de la matriz B.
- Cada operación para realizar dará como producto el elemento de cada posición de una nueva matriz.
- Por cada fila de A multiplicar las columnas de B, en otras palabras: $F_1A \times C_1B$, $F_2A \times C_2B$, $F_nA \times C_nB$.
- La multiplicación se hará elemento a elemento, con sumas consecutivas.
- La posición de las filas y columnas serán la posición del producto generado.
- Conformar la nueva matriz C con los resultados de cada pareja (fila y columna) analizada.

Como resultado de aplicar el algoritmo anterior se tiene lo siguiente:

```
>  #<array '#(3 3) #[36 74 11 11 12 8 143 78 38]>
   #<array '#(3 3) #[32 -56 5 7 -6 4 31 -74 -26]>
   #<array '#(3 3) #[534 2899 376 360 1068 225 514 6129 457]>
```

Nota: se recomienda aplicar el algoritmo de multiplicación por escrito y comparar los resultados.

El resto de las funciones:

```
>  #<array '#(3 3) #[136 36 32 36 12 24 348 8 24]>
   #<array '#(3 3) #[2 2 56 65 9 76 3 2 32]>
   #<array '#(3 3) #[202/309 97/103 451/1236 -3 1/103 6 47/103
   -1 111/206 265/309 -3 29/206 1373/2472]>
   #<mutable-array '#(3 3) #[87 2 6 0 8 19/87 5 19/29 0 0 3
   327/715]>
   '()
```

Ejercicio

Se propone como ejercicio practicar el análisis y manipulación de las funciones descritas en esta sección. Para comprender mejor el concepto de algoritmo se recomienda describir cada paso realizado en la ejecución de alguna de las funciones, como de un código en general.

Extensiones

En este capítulo se explicarán algunas de las librerías proporcionadas por Dr Racket. Aquellas librerías fomentaran temas como interfaces de usuario para desarrollo web, información de proveedor de servicios (host), creación de códigos de identificación QR y de barras.

12. Librerías

12.1. Dr. Racketui

Para implementar diseños gráficos, Dr Racket proporciona estructuras de sintaxis HTML, CSS y Javascript, permitiendo manejar plantillas desde cero. Igualmente contiene librerías alternas como *Dr Racketui* que permiten generar interfaces de usuario sencillas, rápidas y eficaces.

A partir de datos establecidos se crea un formulario web con capacidad para desarrollar una función específica, visualización de menús, guardado de elementos, botones de interacción, conexión directa de servidor, entre otras. Aquellos datos, conocidos como especificaciones web o etiquetas, son las órdenes que inicializan el formulario predispuesto por la librería.

La sintaxis encargada de la elaboración de las interfaces es:

Nota: para cada definición se usa:

[Dato cuerpo (especificación)]

Etiquetas	**Descripción**

`Boolean`
`Number`
`Symbol`
`String`
`String+`
	Datos booleanos, números, símbolos, cadenas.

Etiquetas	Descripción
`(web-launch <nombre del programa> <funciones>)`	Inicializa la interfaz de usuario en un servidor web local.
`Image Filename`	Funciones que reciben o exportan un archivo (fichero).
`(oneof <datos>)`	Escoge un elemento de etiquetas.
`(listof <datos>)`	Crea una lista con las etiquetas proporcionadas.
`(function <propósito(cadena)[<nombre del procedimiento a mostrar> <etiquetas> <+> -> <etiqueta>])`	Representa una función la cual se guía hacia un formulario web, cumpliendo con los parámetros (etiquetas) propuestas. Este se debe proporcionar siempre cuando se quiere mostrar una interfaz de usuario.

Tabla 11. Elaboración de interfaces

Ejemplo

El siguiente código representa la creación de una interfaz de usuario con los elementos de *Dr Racketui:*

```
1   (require Dr Racketui)
2
3   ; acronym : listof string -> string
4   (define (acronym a-los)
5     (cond [(empty? a-los) ""]
6           [(cons? a-los)
7     (if (string-upper-case? (string-ith (first a-
    los) 0))
8             (string-append (string-ith (first a-
    los) 0)
9                                (acronym (rest a-
    los)))
10                (acronym (rest a-los)))]))
11
12  (web-launch
13   "Acronym Builder"
14   (function
15    "Enter some words to build an acronym."
16    (acronym ["Words" (listof+ ["Word" string+])]
17              -> ["The acronym" string])))
```

Luego de instalar la librería *Dr Racketui*, se define el funcionamiento de lo que se mostrara en la web. La función *acronym* recibe una cadena (texto) "*a-los*" que será evaluada por distintas sentencias; si es una cadena vacía se devuelve la representación de una cadena vacía, si la cadena es una pareja de elementos, se evalúa si el primer elemento de la cadena (la primera letra) es mayúscula; si retorna verdadero, la letra se concatena con el siguiente elemento del par anteriormente verificado llamando nuevamente a la función pero esta vez con nuevos elementos.

Las líneas restantes son funciones propias de la librería en uso, *web-launch* permite alojar todo el contenido en un servidor local, resumiendo todo el proceso que conlleva hacerlo manualmente. Se escoge la función modelo a

ejecutar; *function* se encarga de crear una función que reciba cadenas en forma de listas.

Al ejecutar el código se abrirá una pestaña en el navegador web, donde se tendrá una ventana como las que se muestran a continuación:

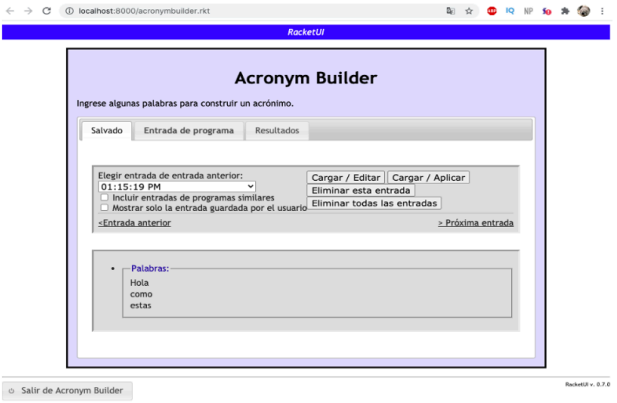


Ilustración 32. Pestaña de interfaz Dr Racketui

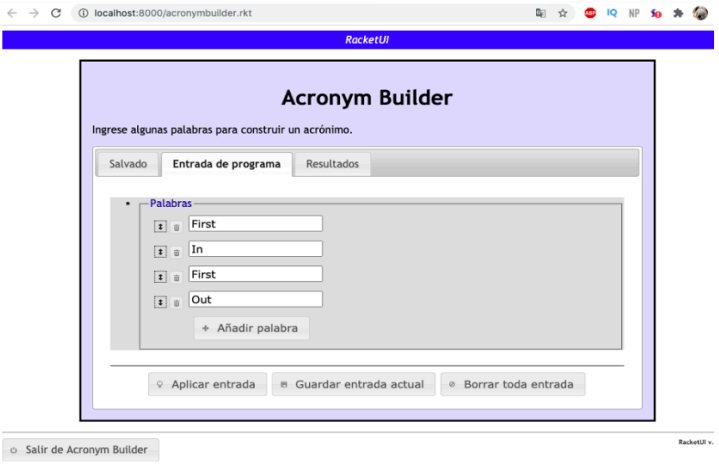


Ilustración 33. Pestaña de interfaz de Dr Racketui

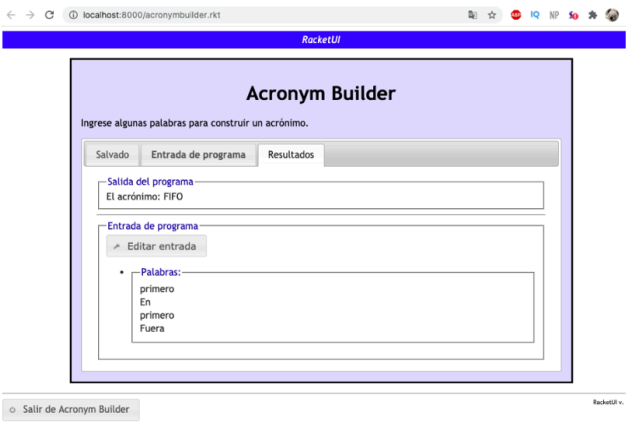


Ilustración 34. Pestaña de interfaz Dr Racketui

12.2 Hostname

El paquete de funciones *hostname* proporciona un conjunto de acciones que determinan información referente al nombre del host y dirección de ip. Es de común uso para acciones con servidores, conexiones a otros dispositivos, pruebas de seguridad de la información, o simplemente conocer la maquina a la cual se maneja.

Las funciones que otorga son:

Funciones	Descripción
`(get-full-hostname)` `(get-short-hostname)`	Retorna el nombre del host (o dominio)completo.

`(get-ipv4-addrs`
`[<#:normal?>`
`<#:localhost?>])`
	Retorna las posibles direcciones para la máquina que está en uso.
	
	Si normal? es verdadero, se devuelven direcciones ip no locales. Caso contrario si localhost? es verdadero.

Tabla 12. Funciones de la libreria Hostname.

Ejemplos.

```
1   #lang Dr Racket
2
3   (require hostname)
4
5   (get-full-hostname)
6
7   (get-short-hostname)
8
9   (get-ipv4-addrs #:normal? #t)
10
11  (get-ipv4-addrs #:localhost? #t)
```

Como resultado se tiene:

```
>  "MacBook-Air-de-Miguel.local"
>  "MacBook-Air-de-Miguel"
>  '("192.168.0.17")
>  '("127.0.0.1" "192.168.0.17")
```

12.3. Generador de códigos de barras y QR

Códigos de barras

La biblioteca *simple-barcode* utiliza dos funciones generales:

Barcode-write: Exporta un archivo que contiene un código de barras. Su estructura es:
`(barcode-write <'tipo de archivo (png o svg)> <código> <nombre del archivo> [<#:code_type (tipos de código como 'ean13, 'code128, 'code39, 'code39_checksum)> <#:color_pair ("black "."white", puede ser cualquier pareja de colores)> <#brick_width (por`

```
defecto 3)> <#:font_size (por defecto
3)>])
```

Barcode-read: Lee un archivo que contiene un código de barras. Su estructura es:

```
(barcode-read <nombre del archivo (o
ruta)> [#:code_type (tipos de código
como ‘ean13, ‘code128, ‘code39,
‘code39_checksum)])
```

Nota: Los tipos de código usados en cada función son distintas formas de distribución de información para la creación de códigos de barras.

Ejemplos

•Escritura

```
1  #lang Dr Racket
2
3  (require simple-barcode)
4
5  (barcode-write 'png "123456789567" "barcodeprueba.png")
```

Como resultado se crea un archivo *.png*:

Ilustración 35. Archivo de código de barras

El archivo almacena el código de barras creado:

Ilustración 36. Código de barras

•Cambios en diseño

Aplicando algunas modificaciones de color y tamaño se obtiene:

```
1  (barcode-
2  write 'png "834567983214" "barcodepruebacolor.png"
   #:color_pair '("white" . "blue"))
```

Ilustración 37. Código de barras con modificación

Nota: Los colores pueden darse en formato hexadecimal, manteniendo la estructura de parejas.

•Lectura

Usando la función de lectura se tiene:

```
1  #lang Dr Racket
2
3  (require simple-barcode)
4
5  (barcode-read "barcodeprueba.png")
```

Como resultado se tiene:

```
>  "1234567895673"
```

Códigos QR

Al igual que con los códigos de barras, la librería *simple-qr* utiliza dos funcionales generales:

Qr-write: Crea un archivo con que contiene el código QR. Su estructura es:

```
(qr-write <ruta guardada en el código
QR> <nombre del archivo (ruta)>
[<#:modo (cadena)>
<#:error_level(cadena)>
<#:module_width (tamaño del código)>
<#:color (“black”.”white”, puede ser
cualquier pareja de colores)>…
<#:output_type (‘png, ‘svg)>]
```

Qr-read: Lee un archivo de códigos QR. Su estructura es:

```
(qr-read <ruta del archivo>)
```

Nota: La ruta almacenada será el sitio al cual se redirecciona luego de escanear dicho código.

Ejemplos

•Escritura

Se tiene el siguiente código:

```
1 #lang Dr Racket/base
2
3 (require simple-qr)
4
5 (qr-write "https://www.youtube.com/"
  "qrprueba.png")
```

Se debe de guardar el código en un archivo, para que al crear el formato QR, esta pueda ser almacenado en la misma ruta del archivo creado; al ejecutarlo, deberá aparecer un documento *.png* como el siguiente:

Ilustración 38. Archivo de código QR

El documento *qrpruebas.png* tendrá almacenado el código QR con la dirección de la página oficial de YouTube:

Ilustración 39. Código QR

Usando una aplicación de lector QR para escanear la imagen, debería ser enviado a la página principal de la plataforma de *youtube.com*.

Aplicando algunas modificaciones de estilos con el código anterior se obtiene:

```
1 (qr-write "https://www.youtube.com/"
  "qrprueba1.png" #:color '("#0FFFC2"."#F40FFF")
2             #:module_width 8)
```

El resultado es:

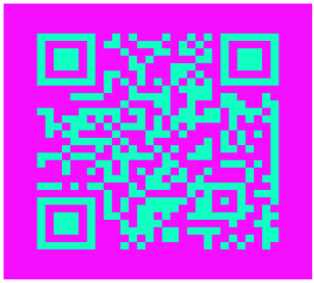


Ilustración 40. Código QR modificado

Nota: Los colores pueden darse en formato hexadecimal, manteniendo la estructura de parejas.

•Lectura

Reutilizando los archivos creados anteriormente, la función *qr-read* retorna la dirección almacenada en el código QR, por ejemplo:

```
1 #lang Dr Racket/base
2
3 (require simple-qr)
4
5 (qr-read "qrprueba.png")
```

Retorna:

```
> "https://www.youtube.com/"
```

13. Referencias

13.1 Bibliografía

Fundamentos y estructuras de datos

NAVAS EDUARDO. (2010). Programando con Racket 5. El Salvador: Universidad Centroamericana “José Simeón Cañas”.

CULPEPPER RYAN.(2017). DB: Database Connectivity, a database interface for functional programmers.

BOTTAZZI CRISTIAN, COSTARELLI SANTIAGO, D’ELIA JORGE, et al. Algoritmos y estructuras de datos.
Argentina: Facultad de ingenierías y ciencias hídricas en Universidad Nacional del Litoral. Centro de
investigación de métodos computacionales.

R. FALL KEVIN, STEVENS W. RICHARD.TCP/IP Illustrated, Volume 1.The Protocols Second Edition

MARTINEZ F. JUAN MANUEL,CASTRO LICEÁGA RAMÓN. (2012). APUNTES DIGITALES PLAN 2012.(México) Universidad Nacional Autónoma de México.

13.2 Sitios de internet para programación con Racket

Racket

https://racket-lang.org/