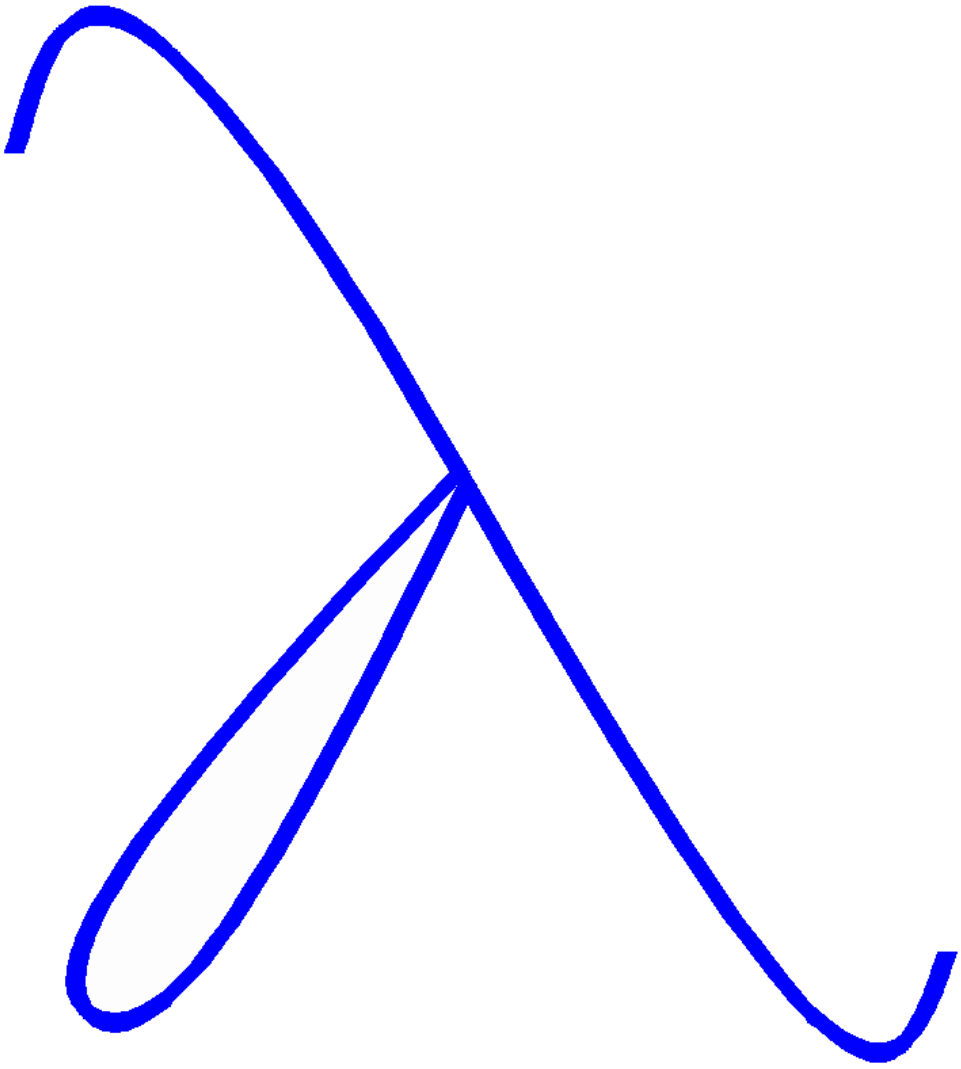
Programando con Racket 5



Eduardo NAVAS

Departamento de Electrónica e Informática,
Universidad Centroamericana
"José Simeón Cañas"

Programando con Racket 5

por *Eduardo NAVAS*

versión 1.0
2010.07.21

Este libro fue desarrollado únicamente con software libre. Entre las herramientas usadas, se encuentran: LaTeX, LyX, GNU/Linux, GNOME, KDE, KmPlot, GIMP, Python, etc.

La versión digital y el material adicional puede ser descargado de:
www.aliamondano-eo.wikidot.com/racket-5
http://dei.uca.edu.sv/publicaciones/

ISBN: 978-99923-73-61-3

Editado y preparado desde el
Departamento de Electrónica e Infomática de la
Universidad Centroamericana "José Simeón Cañas",
El Salvador, Centroamérica.

Dedico esta obra al egoísmo

Prólogo

Este libro evolucionó a partir del material preparado para las clases de la materia Programación Funcional, impartida para la Carrera de Licenciatura en Ciencias de la Computación de la Universidad Centroamericana "José Simeón Cañas".

Después de un año de trabajo, este libro incluye un recorrido por las características básicas del lenguaje Racket, en su versión 5.

Racket 5 es la nueva versión de *PLT Scheme*, un sistema de programación de larga tradición en el aprendizaje de la programación de computadoras, a través del paradigma funcional, basándose en el lenguaje Scheme.

Realmente no existe, formalmente hablando, un lenguaje llamado Scheme, sino que se le llama así a una familia de lenguajes de programación funcionales (véase el capítulo 1).

En este libro, se discute específicamente el dialecto conocido como Racket (anteriormente PLT Scheme), uno de los más difundidos. Si se quiere un estudio más purista sobre Scheme, revise el estándar R5RS que también es soportado por el intérprete de Racket.

Los temas abordados en la Parte I incluyen una introducción a la programación funcional, una sencilla guía de instalación de Racket y una introducción a la interacción con Racket y DrRacket.

En la Parte II se introduce el lenguaje Racket en sí, a través de sus elementos básicos y los bloques lambda, característicos de la programación funcional.

La Parte III describe los demás elementos del lenguaje y contiene múltiples ejercicios para que el lector practique sus nuevos conocimientos.

Finalmente, la Parte IV muestra las capacidades de Racket para implementar programas con interfaces graáficas de usuario.

Y por último, la Parte V incluye un anexo describiendo las diferencias entre la versión 5 de Racket y la serie 4.x de PLT Scheme.

Índice general

Índice de figuras

Parte I

Introducción a la Programación Funcional con Racket

1 Programación Funcional

La programación funcional, iniciada a finales de la década de los 50's, es aquella cuyo paradigma se centra en el Cálculo Lambda. Este paradigma es más útil para el área de inteligencia artificial (ya que satisface mejor las necesidades de los investigadores en esta área), y en sus campos secundarios: cálculo simbólico, pruebas de teoremas, sistemas basados en reglas y procesamiento del lenguaje natural.

La característica esencial de la programación funcional es que los cálculos se ven como una función matemática que hace corresponder entradas y salidas.

1.1. Objetivo

El objetivo es conseguir lenguajes expresivos y matemáticamente elegantes, en los que no sea necesario bajar al nivel de la máquina para describir el proceso llevado a cabo por el programa, y evitando el concepto de estado del cómputo. Los lenguajes funcionales tienen el propósito de acercar su notación a la notación normal de la matemática, cosa que no ocurre, por ejemplo, con los lenguajes imperativos (como *C* o *Java*).

El estado de cómputo o estado de cálculo o estado de programa, se entiende como un registro (con una o más variables) del estado en el que se encuentra el programa en un momento dado. En la práctica, este registro del estado de un programa, se implementa con variables globales, de las que depende el curso de ejecución de alguna parte del programa.

Por ejemplo, considere el siguiente código en lenguaje *C*:

```
1  // Archivo: no-funcional.c
2  #include <stdio.h>
3
4  int variable_contador = 0;
5
6  int aumentar_contador(int incremento){
7      variable_contador += incremento;
8      return variable_contador;
9  }
10
11 void mostrar_contador(void){
12     printf("El valor del contador es: %d\n", variable_contador);
13 }
14
```

```
15 int main(void){
16     mostrar_contador();
17     aumentar_contador(5);
18     mostrar_contador();
19     aumentar_contador(10);
20     mostrar_contador();
21     return 0;
22 }
```

En este pequeño programa, se puede decir que `variable_contador` representa el estado del programa.

1.2. Características

Los programas escritos en un lenguaje funcional están constituidos únicamente por definiciones de funciones, entendiendo éstas, no como subprogramas clásicos de un lenguaje imperativo, sino como funciones puramente matemáticas, en las que se verifican ciertas propiedades como la *transparencia referencial*. La transparencia referencial, significa que *el significado de una expresión depende únicamente del significado de sus subexpresiones o parámetros*, no depende de cálculos previos ni del orden de evaluación de sus parámetros o subexpresiones, y por tanto, implica la carencia total de efectos colaterales. No hay algo como el estado de un programa, no hay variables globales. En el caso del programa `no-funcional.c`, presentado arriba, el resultado de la expresión `aumentar_contador(10)` no sólo depende del número `10`, sino de otra variable ajena a la función.

Otras características propias de estos lenguajes (consecuencia directa de la ausencia de estado de cómputo y de la transparencia referencial) son la no existencia de asignaciones de variables y la falta de construcciones estructuradas como la secuencia o la iteración (no hay `for`, ni `while`, etc.). Esto obliga en la práctica, a que todas las repeticiones de instrucciones se lleven a cabo por medio de funciones recursivas.

1.3. Lenguajes de Programación Funcionales

Existen dos grandes categorías de lenguajes funcionales: *los funcionales puros* y *los funcionales híbridos*. La diferencia entre ambos radica en que los lenguajes funcionales híbridos son menos *dogmáticos* que los puros, al incluir conceptos tomados de los lenguajes imperativos, como las secuencias de instrucciones o la asignación de variables. En contraste, los lenguajes funcionales puros tienen una mayor potencia expresiva, conservando a la vez su transparencia referencial, algo que no se cumple siempre con un lenguaje funcional híbrido. Sin embargo, es de mencionar que en un lenguaje de programación funcional puro, en la práctica, sería muy difícil programar sistemas; aunque son muy buenos para aplicaciones eminentemente matemáticas.

Entre los lenguajes funcionales puros, cabe destacar a Haskell y Miranda. Los lenguajes funcionales híbridos más conocidos son Lisp, los dialectos de Scheme y Ocaml. Erlang es un lenguaje funcional de programación concurrente. R es un lenguaje funcional dedicado a la estadística. Mathematica y Maxima son también lenguajes/entornos funcionales, orientados totalmente al álgebra simbólica.

Entre otros lenguajes que se podrían utilizar para programación funcional, se podrían incluir a Perl, usando exclusivamente funciones definidas por el usuario. Así como Python, como lenguaje que incorpora el paradigma funcional.

1.4. Ejemplos de código de lenguajes funcionales

En Haskell:

```
1     --Función recursiva para calcular el factorial de un número
2     factorial :: Integer -> Integer
3     factorial n = if n==0 then
4                           1
5                   else
6                           n * factorial (n - 1)
7
8     --Sumar elementos de una lista
9     sumar :: [Integer] -> Integer
10    sumar [] = 0
11    sumar (x:xs) = x+sumar(xs)
12
13    --Función para calcular el valor de e (2.71828182845905)
14    euler :: Double -> Double
15    euler 0.0 = 1.0
16    euler n   = 1.0 / product [1..n] + euler (n - 1.0)
```

En Miranda:

```
1   //Lista de cuadrados de n donde n es tomado de la lista de todos los
        enteros positivos:
2   squares = [ n * n | n <- [1..] ]
```

En OCaml:

```
1   (* Longitud de una lista *)
2   let rec long  = function
3     |[] -> 0
4     |x::xs -> 1 + long xs;;
```

En Erlang:

```
1   fac(0) -> 1;
2   fac(N) when N > 0 -> N * fac(N-1).
```

En Python:

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [[x,x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]

>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
```

2 Instalación de Racket

2.1. Instalación con el instalador oficial

A continuación de describen los pasos básicos de instalación:

1. Vaya al sitio http://racket-lang.org/download/ y descargue el instalador correspondiente o más cercano a su distribución, tal como se muestra en la figura 2.1.

2. El archivo descargado es un **.sh**, por lo que hay que asignarle los permisos de ejecución, necesarios para poder ejecutarlo[1]:
   ```
   $ chmod u+x racket-xxxx.sh
   $ ./racket-xxxx.sh
   ```

3. A continuación, el instalador pregunta si se desea hacer una instalación tipo Unix o una instalación en una sóla carpeta. La opción por defecto es no hacer una instalación tipo Unix.

4. Luego se pregunta en cuál carpeta se desea realizar la instalación (en caso de haber respondido "no" en el paso anterior. La opción por defecto es `/usr/plt`, para la cual se requiere tener permisos de superusuario. También se puede instalar en la carpeta del usuario o en cualquier otra.

5. A continuación se procede a realizar la instalación en la carpeta elegida y aquí termina la instalación.

Automáticamente se instalan las páginas de la documentación oficial de Racket en la carpeta de instalación elegida. Si la carpeta de instalación elegida no fue en la carpeta del usuario, es muy probable que la carpeta de la documentación sea `/usr/share/plt`, `/usr/share/doc/plt`, o `/usr/plt/doc/`. En esta carpeta habrá un archivo `index.html`. En todo caso, con el comando

```
$ raco docs
```

se lanza la página del índice de la documentación instalada con el navegador por defecto del Sistema Operativo.

[1] o simplemente ejecutar:
```
$ sh racket-xxxx.sh
```

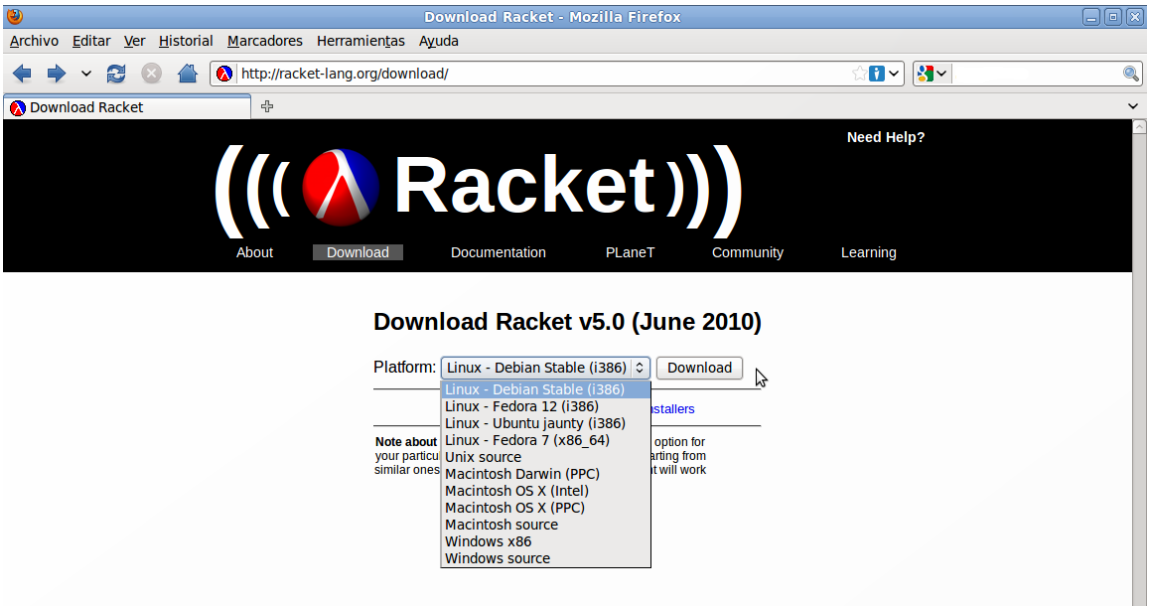


Figura 2.1: Sitio de descarga de Racket

2.2. Instalación desde repositorios

Por el momento[2], las distribuciones de GNU/Linux no incluyen la nueva versión (la 5.0) de *Racket*. En su lugar, todavía incluyen la serie 4.x de *PLT Scheme* (que es completamente compatible con todo el contenido de este libro[3], en esta primera versión). Es sólo cuestión de tiempo (unos 6 meses o un año) para que la nueva versión de *PLT Scheme*, llamada *Racket*, se encuentre en los repositorios de las distribuciones más difundidas.

El proceso de instalación es en general muy sencillo. Especialmente cuando usamos una distribución de GNU/Linux que contiene a *PLT Scheme* en sus repositorios.

2.2.1. Debian

En distribuciones basadas en *Debian*, basta con instalar el paquete `plt-scheme`:

```
# apt-get install plt-scheme
```

También sería buena idea, instalar el paquete `plt-scheme-doc` que instala la documentación oficial de la versión instalada por el paquete anterior. Con este paquete, está disponible la página `/usr/share/plt/doc/index.html` que es el índice de la documentación.

[2] Al momento de escribir este libro
[3] véase el apéndice A

Para instalar la documentación junto con el programa, ejecute el siguiente comando:
`# apt-get install plt-scheme plt-scheme-doc`

2.2.2. Fedora

En distribuciones *Fedora*, basta con instalar el paquete `plt-scheme`:
`# yum install plt-scheme`

En esta distribución no se encuentra la documentación como paquete, por lo que hay que consultarla en línea, o descargarla.

3 Expresiones Racket - Notación Prefija

3.1. Notación para la sintaxis de Racket

Desde este momento en adelante, se utilizará la siguiente notación tipo BNF para explicar la sintaxis del lenguaje:

- Todas las secuencias de caracteres delimitadas por `<` y `>` representan símbolos no terminales. Por ejemplo: `<símbolo_no_terminal>`.
- Todas las secuencias de caracteres no delimitadas, representan símbolos terminales. Por ejemplo: `define`, `(`, `)`, `let`.
- El metaagrupamiento se hace con llaves: `{` y `}`.
- El metasímbolo `+`, indica al menos una ocurrencia del símbolo precedente.
- El metasímbolo `*`, indica ninguna, una o varias ocurrencias del símbolo precedente.

3.2. Notación prefija de Racket

En Racket, todas las expresiones tienen la forma: `(<operador> <operando>*)`, es decir, que están siempre en notación prefija con pareamiento completo:
`(* 2 3) ->` equivale a `(2 * 3)`
`(> 5 6)->` equivale a `(5 > 6)`
`(+ 2 3 10)->` equivale a `(2 + 3 + 10)`
`(+ 4 (* 3 2))->` equivale a `(4 + 3 * 2)`

Por ejemplo, la expresión infija $5a + 2bc^2$ es: `(+ (* 5 a) (* 2 b c c))`.

4 Interacción con Racket

Dependiendo de cómo se vea, Racket es:

- un lenguaje de programación,
- una familia de lenguajes de programación, variantes de Scheme, que a su vez, es un dialecto de Lisp; o
- un conjunto de herramientas de programación.

Racket tiene básicamente dos herramientas principales:

- `racket`, el compilador, intérprete y sistema de ejecución interactiva; y
- DrRacket, un IDE que corre sobre `racket` (es decir, que lo usa como motor de ejecución y compilación).

En el caso de **DrRacket**[1], debe especificarse el lenguaje en particular que se va a utilizar, ya que este Entorno se acomoda a diversas variantes de Scheme soportadas por el intérprete y compilador **racket**. En nuestro caso particular, usaremos la opción "Usar el lenguaje declarado en el código fuente".

Cuando se selecciona esta opción, en el área de texto para escribir el programa, aparece la línea:

```
#lang racket
```

Esta línea, al inicio de cualquier archivo de texto, indica que el código a continuación, es la variante más completa (en términos de bibliotecas disponibles y capacidad del lenguaje) de todas las variantes soportadas por Racket, conocido como *Lenguaje Racket*.

Cuando se ejecuta el Racket en la línea de comandos (con el comando `racket`), el lenguaje por omisión es esta variante.

4.1. Ejecución interactiva

La parte de abajo de la interfaz de DrRacket (comando `drracket`), y la herramienta de línea de comandos `racket`, funcionan como un área de interacciones, al estilo de una terminal normal. En ella, se puede escribir una expresión (aquí no hay comandos, ni instrucciones), presionar Intro y el intérprete devuelve el resultado de la expresión.

[1] Para mayor información sobre DrRacket, léase la documentación oficial de DrRacket.

Por ejemplo:

```
> 5
5
> (+ (sqr 4) (sqr 3))
25
> "¡Hola Mundo!"
"¡Hola Mundo!"
> (substring "¡Hola Mundo!" 6 11)
"Mundo"
```

En la última expresión, se invocó a una función llamada `substring`, con tres parámetros: una cadena, y dos números enteros.

4.1.1. Definiciones e interacciones con DrRacket

Se pueden definir funciones propias, basándose en otras funciones como `substring`. Para ello, en el área de definiciones (el área de texto superior) se escribe algo como:

```
#lang racket
;extraer.rkt

(define (extraer str)
  (substring str 4 7)
  )
```

Presionar el botón Run y luego, en el área de interacciones (la terminal de abajo), ya se puede invocar esa función:

```
> (extraer "1234567890")
"567"
> (extraer "este es un texto con muchos caracteres")
" es"
```

4.1.2. Ejecución interactiva con Racket

Para poder hacer esto mismo con `racket`, primero hay que guardar las definiciones en un archivo (por convención, con extensión **`.rkt`**), por ejemplo en un archivo llamado `extraer.rkt`. Entonces, en la línea de comandos, hacemos:

```
> (enter! "extraer.rkt")
> (extraer "1234567890")
"567"
```

La función `enter!` carga el archivo pasado como parámetro y cambia el contexto de evaluación a las definiciones del archivo, igual que el botón `Run` de DrRacket.

4.2. Ejecución no interactiva

Si tiene el archivo:

```
1 #lang racket
2 ;extraer2.rkt
3
4 (define (extraer str)
5   (substring str 4 7)
6   )
7
8 (extraer "1234567890")
```

Ese es un programa completo que imprime en pantalla "567" cuando se ejecute.

Para ejecutarlo, vaya la línea de comandos:

```
1 $ racket extraer2.rkt
2 "567"
3 $
```

Se dice que es ejecución no interactiva porque uno no puede invocar a voluntad funciones definidas en el archivo. Sólo se ejecutan las definiciones y llamadas que se encuentran en el archivo. Sin embargo, los programas pueden ser interactivos en el sentido que el curso de la ejecución se puede cambiar en tiempo de ejecución.

4.2.1. Parámetros de la línea de comandos

La ejecución de un programa, puede controlarse, por ejemplo, con parámetros de la línea de comandos. Esto se logra con la función `current-command-line-arguments` que retorna un vector inmutable de cadenas inmutables, una por cada parámetro. Por ejemplo, considere el siguiente programa:

```
1 #lang racket
2 ;linea-de-comandos.rkt
3
4 (display "Los parámetros pasados al programa son: \n")
5 (write (current-command-line-arguments))
6 (newline)
```

Puede ejecutarlo así:

```
$ racket linea-de-comandos.rkt hola, "esta es" una prueba
```

Y la salida sería:

```
Los parámetros pasados al programa son: #("hola," "esta es" "una" "prueba")
```

5 Compilación de programas Racket

5.1. Lo básico sobre compilación

Como parte de Racket, se incluye el programa `raco` que es la herramienta de compilación de Racket. Puede obtener una breve descripción de todas sus posiblidades con `$ raco --help`.

La compilación de un programa Racket es muy sencilla. Suponga que tiene el programa:

```
1 #lang racket
2 ;hola.rkt
3 (display "¡Hola Mundo!\n")
```

Se compila así:
`$ raco exe -o ejecutable hola.rkt`

Para ejecutarlo, dependiendo de la configuración de la terminal:

```
1 $ ./ejecutable
2 ¡Hola Mundo!
3 $
```

5.2. Compilación con múltiples módulos

El hecho de tener un programa separado en módulos funcionales, no afecta el proceso de compilación. Simplemente hay que compilar el archivo que contiene la ejecución inicial de nuestra aplicación. Por ejemplo, considere los siguientes dos archivos:

```
1  #lang racket
2  ;principal.rkt
3
4  (require "secundario.rkt")
5
6  (define parámetros (current-command-line-arguments))
7
8  (función-pública "hola")
9  (función-pública constante-de-módulo)
10 (if ((vector-length parámetros) . > . 0)
11     (for-each función-pública (vector->list parámetros))
12     (display "No se pasaron parámetros\n")
13     )
14 (display "¡Adiós!\n")
```

```
1  #lang racket
2  ;secundario.rkt
3
4  (provide función-pública constante-de-módulo)
5
6  (define constante-de-módulo "Esta constante está en secundario.rkt")
7
8  (define (función-pública parámetro)
9    (función-privada parámetro)
10   )
11
12 (define (función-privada parámetro)
13   (display "Esta es una función declarada e implementada en secundario.rkt\
         n")
14   (display "El parámetro pasado es: ")
15   (write parámetro)
16   (newline)
17   )
18
19 "Cuando se importa un módulo, se ejecuta como un script"
```

Y para compilarlos, simplemente se hace:

```
$ raco exe -o ejecutable principal.rkt
```

Y para ejecutar el programa:

```
$ ./ejecutable "otros parámetros" unidos
"Cuando se importa un módulo, se ejecuta como un script"
Esta es una función declarada e implementada en secundario.rkt
El parámetro pasado es: "hola"
Esta es una función declarada e implementada en secundario.rkt
El parámetro pasado es: "Esta constante está en secundario.rkt"
Esta es una función declarada e implementada en secundario.rkt
El parámetro pasado es: "otros parámetros"
Esta es una función declarada e implementada en secundario.rkt
El parámetro pasado es: "unidos"
¡Adiós!
$
```

Parte II

Introducción al lenguaje Racket

6 Elementos básicos

A continuación se presenta un recorrido por las principales y más básicas partes del lenguaje Racket.

6.1. Comentarios

Los comentarios son elementos escenciales en todo lenguaje de programación, ya que permiten que el programador aclare la futura lectura del código fuente.

En Racket, los comentarios de una línea comienzan con ; y los comentarios de bloque son delimitados por `#|` y `|#`.

Ejemplo:

```
1 #lang racket
2 ;Todos los programas Racket deben comenzar con esta línea de arriba.
3
4 #|
5     Este es un comentario en Racket,
6     que tiene varias líneas.
7 |#
8 (display "Bueno, esto no es comentario, es código\n") ;;Esto sí ;-)
```

6.2. Definiciones Globales

Una definición de la forma
`(define <identificador> <expresión>)`
le asigna a `<identificador>` el resultado de evaluar `<expresión>`.

Una definición de la forma
`(define (<identificador> <identificador>*) <expresión>+ )`
le asigna al primer `<identificador>` una función (o procedimiento) que toma tantos argumentos como `<identificador>`es restantes haya dentro de los paréntesis. El cuerpo de la función es la serie `<expresión>+` y cuando la función es llamada, devuelve el resultado de la última `<expresión>`.

Ejemplo:

```
(define máximo 3)           ;Define que máximo es 3
(define (prefijo str)       ;Define que prefijo es una función de un
    argumento
  (substring str 0 máximo)
  )

> máximo
3
> (prefijo "Hola, ¿cómo estás?")
"Hol"
> prefijo
#<procedure:prefijo>
> substring
#<procedure:substring>
```

Una función puede tener múltiples expresiones, pero la función sólo devuelve el resultado de la última:

```
(define (hornear sabor)
  (printf "Precalentando el horno para...\n")
  "Esta cadena es completamente ignorada"
  (string-append "pastel de " sabor)
  )

> (hornear "manzana")
Precalentando el horno para...
"pastel de manzana"
```

6.2.1. Identificadores

Los identificadores en Racket son muy liberales. A diferencia de otros lenguajes de programación que restringen mucho los caracteres válidos para sus identificadores, en Racket, prácticamente no hay restricciones.

Los únicos caracteres no válidos en los identificadores son: `( ) [ ] { } " , ' ` ; # | \`. Tampoco se pueden utilizar identificadores que se correspondan con literales numéricos y tampoco están permitidos los espacios dentro de los identificadores.

Por lo demás, se pueden utilizar identificadores como `"variable-con-guiones"`, `"variable+con+más-y-`
`"123abc"`, `"+-"`, `"¿variable-interrogativa???"`, "`23..4`", "`variable/dividida`", etc.

Y, puesto que, el intérprete racket procesa archivos Unicode, los identificadores pueden contener y estar formados por cualesquiera caracteres válidos en esa codificación (se recomienda que los archivos de código fuente estén en codificación UTF-8). Por ejemplo, las siguientes declaraciones son válidas para el intérprete de Racket:

```
> (define áéíóúü-en-español 1)
> (define üäöß-deutsch 2)
> (define eĥoŝanĝo-ĉiuĵaŭde-en-Esperanto 3)
```

6.3. Llamadas a funciones

Típicamente una llamada a función tiene la forma
`(<identificador> <expresión>*)`
donde la secuencia de expresiones determina el número de parámetros reales pasados a la función referenciada por el identificador. Por ejemplo `(prefijo "hola")` o `(hornear "piña")`.

Racket define muchas funciones integradas del lenguaje. Por ejemplo `string-append`, `substring`, `string-length`, `string?`, `sqrt`, `+`, `-`, `<`, `>=`, `number?`, `equal?`, etc.

6.4. Bloques condicionales

6.4.1. `if`

La forma de un bloque condicional en Racket es:
`(if <expresión-lógica> <expresión-para-verdadero> <expresión-para-falso>)`

Cuando se evalúa un `if`, se evalúa la primera expresión. Si esta resulta verdadera, se retorna el resultado de la segunda expresión, y de lo contrario, el resultado de la tercera.

Por ejemplo:

```
> (if (< 1 2)
    "menor"
    "mayor")
"menor"

> (if (positive? (sqrt 4)) "sí es positivo" "no es positivo")
"sí es positivo"

> (define (responder-saludo s)
    (if (equal? "hola" (substring s 0 4))
        "¡hola, gusto de verte!"
        "¿perdón?"
        )
    )

> (responder-saludo "hola programa")
"¡hola, gusto de verte!"

> (responder-saludo "El día está muy bonito, ¿verdad?")
"¿perdón?"
```

Como en otros lenguajes de programación, las sentencias condicionales (así como muchas otras cosas) se pueden anidar dentro de otras:

```
> (define (responder-saludo s)
    (if (string? s)
        (if (equal? "hola" (substring s 0 4))
            "¡hola, gusto de verte!"
            "¿perdón?"
            )
        "perdón, ¿qué?"
        )
    )
> (responder-saludo "hola programa")
"¡hola, gusto de verte!"
> (responder-saludo 3.1416)
"perdón, ¿qué?"
> (responder-saludo "El día está muy bonito, ¿verdad?")
"¿perdón?"
```

Esto también se podría escribir como:

```
> (define (responder-saludo s)
    (if (if (string? s)
            (equal? "hola" (substring s 0 4))
            #f)
        "¡hola, gusto de verte!"
        "perdón, ¿qué?"
        )
    )
```

6.4.2. `and` y `or`

En Racket, las funciones lógicas de conjunción y disyunción, son respectivamente `and` y `or` y su sintaxis es: `(and <expresión>*)` y `(or <expresión>*)`.

La primera retorna `#f` si encuentra que uno de sus parámetros se evalúa a `#f`, y retorna `#t` en caso contrario. La segunda retorna `#t` si encuentra que uno de sus parámetros se evalúa a `#t` y retorna `#f` en caso contrario. Funcionan como se espere que funcionen en otros lenguajes de programación, y además funcionan en cortocircuito (como en otros lenguajes como *C* y *Java*).

Ejemplo:

```
> (and (< 3.1416 (expt 10.1424 3.8))
       (not (negative? pi)))
#t

> (define (responder-saludo s)
    (if (and (string? s)
             (>= (string-length s) (string-length "hola"))
             (equal? "hola" (substring s 0 4)))
        "¡hola, gusto de verte!"
```

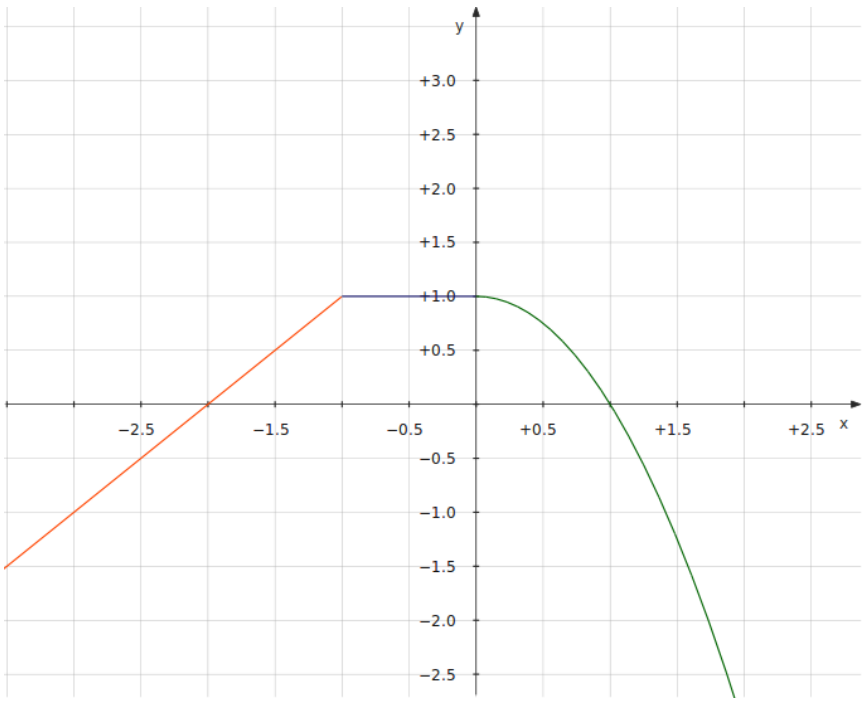


Figura 6.1: Gráfica de ecuación seccionada $f(x) = \begin{cases} x+2 & x < -1 \\ 1 & -1 \leq x < 0 \\ -x^2+1 & 0 \leq x \end{cases}$

```
10          "perdón, ¿qué?"
11          )
12      )
```

6.4.3. `cond`

Una forma de bloques condicionales anidadas (if anidados) es:
`(cond { [ <expresión-de-prueba> <expresión>* ] }* )`

Este bloque condicional contiene una secuencia de cláusulas entre corchetes. En cada cláusula, la primera expresión es una expresión de prueba o evaluación. Si esta se evalúa a verdadero, entonces las restantes cláusulas del grupo son evaluadas, y la sentencia completa retorna el valor de la última expresión de esa cláusula; el resto de las cláusulas son ignoradas.

Si la evaluación de la expresión de prueba se evalúa a falso, entonces el resto de las expresiones de la cláusula son ignoradas y la evaluación continúa con la próxima cláusula. La última cláusula puede usar la constante else que es un sinónimo para `#t`.

Por ejemplo (ver la figura 6.1):

```
> (define (seccionada x)
    (cond [(< x -1)                  ;   x<-1: x+2
           (+ x 2)]
          [(and (>= x -1) (< x 0))   ;-1<x<0 : 1
           1]
          [(>= x 0)                  ; 0<x   : -x^2+1
           (+ (- (sqr x)) 1)]))

> (seccionada -4)
-2
> (seccionada -.5)
1
> (seccionada 1)
0

>  (define (responder-más s)
     (cond
       [(equal? "hola" (substring s 0 4))
        "¡hola, gusto de verte!"]
       [(equal? "adiós" (substring s 0 5))
        "¡nos vemos, que te vaya bien!"]
       [(and (equal? "¿" (substring s 0 1))
             (equal? "?" (substring s (- (string-length s) 1))))
        "No sé"]
       [else "perdón, ¿qué?"]))

> (responder-más "¿hoy?")
"No sé"
> (responder-más "hola pepe")
"¡hola, gusto de verte!"
> (responder-más "la derivada de la función exponencial es ella misma")
"perdón, ¿qué?"
> (responder-más "adiós programa")
"¡nos vemos, que te vaya bien!"
```

En Racket, el uso de paréntesis y corchetes es completamente intercambiable, mientras un (se cierre con un) y un [se cierre con un] no hay problema. Sin embargo, el uso de corchetes junto a los paréntesis hace del código Racket ligeramente más legible.

6.4.4. `case`

Los bloques `case` sirven para corresponder el resultado de una expresión con una serie de valores y evaluar diferentes expresiones en función de eso. La sintaxis básica es:
`(case <expresión-de-prueba> { [ ( <valores>+ ) <expresión>+ ] }* )`

Por ejemplo:

```
> (case (+ 7 5)
```

```
    [(1 2 3) "pequeño"]
    [(10 11 12) "grande"])
"grande"
> (case (- 7 5)
    [(1 2 3) "pequeño"]
    [(10 11 12) "grande"])
"pequeño"
> (case (* 7 5)
    [(1 2 3) "pequeño"]
    [(10 11 12) "grande"]
    [else "fuera de rango"])
"fuera de rango"
```

6.5. Bloques de código secuencial

En la idea básica del paradigma funcional, no existe algo como la *secuencia de instrucciones*, pero como Racket es híbrido, sí disponde esta característica. La secuencia de instrucciones está presente de manera nativa en los bloques `lambda`, `define` (para funciones), `cond`, `case` y `let`, por lo que esas alternativas suelen bastar. Pero para aquellos casos en los que no, se dispone del bloque `begin`:

```
> (if (< 5 6)
      (begin
        (display "Aquí podemos escribir muchas cosas\n")
        "Pero sólo el último elemento será el resultado"
        "cinco es menor que seis"
        )
      "cinco no es menor que seis"
      )
Aquí podemos escribir muchas cosas
"cinco es menor que seis"
```

6.6. Más sobre llamadas a funciones

Racket es un lenguaje muy potente y muy expresivo. Las llamadas a funciones, no sólo pueden hacerse utilizando directamente los identificadores de las funciones. También pueden hacerse utilizando expresiones que devuelvan referencias a funciones. Así, la sintaxis de llamadas a funciones se puede ampliar[1] como:
`(<expresión-de-función> <expresión>* )`

La `<expresión-de-función>`, debe ser una expresión cuyo resultado sea *una función*.

Por ejemplo:

[1] Esta aún no es la forma más general

```
> (define (duplicar valor)
    ((if (string? valor) string-append +) valor valor))

> (duplicar "cadena")
"cadenacadena"
> (duplicar 3)
6
```

Aquí, el bloque `if` retorna una función a través de su nombre (`string-append` o `+`).

Si la `<expresión-de-función>` no devolviera una función, se generaría un error, ya que el primer elemento dentro de los paréntesis debe ser una función. Por ejemplo, la siguiente expresión:
`> (1 2 3)`

produce el error:
`procedure application: expected procedure, given: 1; arguments were: 2 3`

Note que, puesto que una función puede ser devuelta por una expresión, una función también puede se pasada como parámetro a otra función:

```
> (define (componer función valor)
(función (función valor)))
> (componer sqrt 256)
4
> (componer sqr 2)
16
```

7 Funciones anónimas - Bloques `lambda`

Considere la siguiente expresión:
`(+ 5 4)`

Es equivalente a:

```
(define a 5)
(define b 4)
...
(+ a b)
```

La segunda forma sería innecesariamente larga si los valores de a y b sólo se utilizarán una vez. De la misma manera, cuando una función sólo se llama una vez, tener que declararla es innecesariamente largo. Por ello, Racket incluye la posibilidad de escribir funciones anónimas.

Por ejemplo:

```
> (define (poner-admiración s)
    (string-append "¡" s "!"))
> (componer poner-admiración "hola")
"¡¡hola!!"
```

Pero suponga que la función `poner-admiración` sólo llamará cuando se llame una vez a `componer`. Entonces puede escribir la función `poner-admiración` directamente en la llamada a `componer` desde donde será invocada. Entonces se usan los bloques `lambda`.

7.1. Bloques `lambda`

En Racket –así como en muchos otros lenguajes de programación–, un **bloque lambda** produce una función directamente, sin tener que declararla.

El bloque lambda tiene la siguiente sintaxis:
`(lambda ( <identificador>* ) <expresión>+ )`

La serie de identificadores se corresponde, uno a uno, con los parámetros formales de la función a producir; y las expresiones son el cuerpo de la función. Como en la declaración

tradicional de funciones (con `define`)[1], el resultado de la función (cuando se llame), es el resultado de la última expresión del cuerpo del bloque `lambda`.

La evaluación de un bloque `lambda`, produce en sí misma una función:

```
> (lambda (s) (string-append "¡" s "!"))
#<procedure>
```

Entonces, usando `lambda`, la llamada a componer puede ser reescrita como:

```
> (componer (lambda (s) (string-append "¡" s "!")) "hola")
"¡¡hola!!"
> (componer (lambda (s) (string-append "¿¡" s "!?")) "hola")
"¿¡¿¡hola!?!?"
```

7.2. Funciones/Expresiones que producen funciones

Otro uso de `lambda` es como resultado para una función (o expresiones) que produce funciones:

```
> (define (hacer-Agregar-afijos prefijo sufijo)
    (lambda (s) (string-append prefijo s sufijo)))

> (componer (hacer-Agregar-afijos "<" ">") "hola")
"<<hola>>"
> (componer (hacer-Agregar-afijos "¡" "!") "hola")
"¡¡hola!!"
> (componer (hacer-Agregar-afijos "<<" ">>") "hola")
"<<<<hola>>>>"
```

También pueden asignarse el resultado de una función que retorna funciones a un identificador:

```
> (define poner-admiración (hacer-Agregar-afijos "¡" "!"))
> (define menos-seguro (hacer-Agregar-afijos "¿" "?!"))
> (componer menos-seguro "ah nombre")
"¿¿ah nombre?!?!"
> (componer poner-admiración "en serio")
"¡¡en serio!!"
```

También puede asignarse directamente un bloque `lambda` a un identificador. Las siguientes dos definiciones son equivalentes:

```
> (define (poner-admiración s)
    (string-append "¡" s "!"))

> (define poner-admiración
```

[1] En realidad, lo tradicional, es usar `lambda` para definir funciones.

```
5       (lambda (s)
6         (string-append "¡" s "!")))
7
8   > poner-admiración
9   #<procedure:poner-admiración>
```

8 Asignación local

Hay al menos tres formas de hacer asignación local en Racket: Con `define`, con `let` y con `let*`.

8.1. `define`

Hagamos otra ampliación de la sintaxis para los bloques de funciones:
`(define (<identificador> <identificador>* ) <definición>* <expresión>+ )`
y
`( lambda ( <identificador>* ) <definición>* <expresión>+ )`

La diferencia con respecto a la sintaxis anteriormente mostrada, es que hay un bloque opcional de definiciones antes del cuerpo de la función. Por ejemplo:

```
1  > (define (conversar s)
2      (define (¿comienza-con? prefijo) ; local a conversar
3        (define longitud-prefijo (string-length prefijo))  ; local a
             ¿comienza-con?
4        (and (>= (string-length s) longitud-prefijo)
5             (equal? prefijo (substring s 0 longitud-prefijo))))
6      (cond
7        [(¿comienza-con? "hola") "hola, ¿qué ondas?"]
8        [(¿comienza-con? "adiós") "adiós, nos vemos"]
9        [else "¿ah?"]))
10
11 > (conversar "hola programa")
12 "hola, ¿qué ondas?"
13 > (conversar "hace frío en los talleres")
14 "¿ah?"
15 > (conversar "adiós programa")
16 "adiós, nos vemos"
17 > ¿comienza-con?
18 reference to an identifier before its definition: ¿comienza-con?
```

Todas las definiciones dentro de la definición de una función, son locales a ella, y por tanto, invisibles desde fuera de ella. Como todo en Racket, las definiciones se pueden anidar indefinidamente unas dentro de otras.

8.2. `let`

Otra forma de hacer asignaciones locales, es con el bloque `let`. Una ventaja de `let` sobre `define` es que puede ser colocada en cualquier lugar dentro de una expresión y no sólo al principio de la función, como `define`. Además, con `let` se pueden hacer múltiples asignaciones al mismo tiempo, en lugar de hacer un `define` para cada asignación.

La sintaxis de `let` es:
`(let ( { [<identificador> <expresión>] }* ) <expresión>+ )`

Cada cláusula de asignación es un `<identificador>` y una `<expresión>` rodeadas por corchetes, y las expresiones que van después de las cláusulas, son el cuerpo del `let`. En cada cláusula, al `<identificador>` se le asigna el resultado de la `<expresión>` para ser usado dentro del cuerpo. Fuera del bloque `let`, los identificadores no son visibles.

Por ejemplo:

```
1  > (let ([x 1]
2          [y 2])
3      (display (string-append "La suma de "
4                              (number->string x)
5                              " más "
6                              (number->string y)
7                              " es: "
8                              (number->string (+ x y)))))
9  La suma de 1 más 2 es: 3
10 > (+ x y)
11 reference to an identifier before its definition: x
```

8.3. `let*`

Las asignaciones de `let` están disponibles sólo en el cuerpo del `let`, así que las cláusulas de asignación no se pueden referir unas a otras. El bloque `let*`, por el contrario, permite que cláusulas posteriores, referencien cláusulas anteriores:

```
1  > (let* ([x 1]
2           [y 2]
3           [z (+ x y)])
4      (printf "La suma de ~a y ~a es: ~a" x y z))
5  La suma de 1 y 2 es: 3
```

Parte III

Elementos del lenguaje

9 Listas e Iteración

En este capítulo se describen los pares y sus casos particulares, las listas. Además, se describen los mecanismos propios de Racket para procesar y recorrer listas.

9.1. Listas

Las listas son el tipo de dato más prominente de Racket, como dialecto de Scheme y a su vez de Lisp. No es de extrañar que haya funciones especialmente avanzadas y de alto nivel para procesar y manipular listas.

Hay varias maneras diferentes de crear listas en Racket. La principal de ellas es utilizando la función `list`:

```
> (list "rojo" "verde" "azul")
("rojo" "verde" "azul")

> (list 1 2 3 4)
(1 2 3 4)

> (list (exp 1) (sqrt 2))
(2.718281828459045 1.4142135623730951)

> (list "cadena" 123 9.87654)
("cadena" 123 9.87654)

> (define mi-lista (list "a" 2 3.1416))

> mi-lista
("a" 2 3.1416)
```

Otra forma, es utilizar la notación tradicional de Lisp, con apóstrofe:

```
> '("otra lista" "con números" 3 4 5.322)
("otra lista" "con números" 3 4 5.322)

> (define mi-lista '("otra lista" "con números" 3 4 5.322))

> mi-lista
("otra lista" "con números" 3 4 5.322)
```

9.1.1. Lista vacía o nula

La lista vacía, se puede escribir de diversas maneras en Racket:

- Invocando a la función `list` sin parámetros: `(list)`
- Con la constante especial `empty`
- Con la constante especial `null`
- Con la forma tradicional de Lisp: `'()` , que es un caracter de apóstrofe seguido de paréntesis vacíos.

Para verificar si una expresión se evalúa a una lista vacía, se pueden utilizar también varias funciones:

- La función de evaluación lógica `empty?`: `(if (empty? L) "vacía" "no vacía")`
- La función `null?`: `(if (null? L) "vacía" "no vacía")`

Ejemplos:

```
> '()
()

> empty
()

> null
()

> (list)
()

> ((lambda (L) (if (empty? L) "vacía" "no vacía")) null)
"vacía"

> ((lambda (L) (if (empty? L) "vacía" "no vacía")) '() )
"vacía"

> ((lambda (L) (if (null? L) "vacía" "no vacía")) empty)
"vacía"

> ((lambda (L) (if (null? L) "vacía" "no vacía")) '("a"))
"no vacía"
```

9.1.2. Funciones básicas sobre listas

- `length` para verificar la longitud de una lista
- `list-ref` para extraer el i-ésimo elemento de una lista (los índices comienzan desde cero, como en la mayoría de lenguajes de programación).

- `append` para unir listas
- `reverse` para invertir el orden de una lista
- `member` para verificar si un elemento está en una lista
- `list?` para verificar si un identificador se corresponde con una lista (que puede estar vacía)

Ejemplos:

```
> '("cero" 1 "dos" 3 "cuatro" 5.322)
("cero" 1 "dos" 3 "cuatro" 5.322)

> (length '("cero" 1 "dos" 3 "cuatro" 5.322))
6

> (list-ref '("cero" 1 "dos" 3 "cuatro" 5.322) 2)
"dos"

> (list-ref '("cero" 1 "dos" 3 "cuatro" 5.322) 5)
5.322

> (list-ref '("cero" 1 "dos" 3 "cuatro" 5.322) 6)
list-ref: index 6 too large for list: ("cero" 1 "dos" 3 "cuatro" 5.322)

> (append '("cero" 1 "dos" 3 "cuatro" 5.322) (list "a" "b" "c") (list "un
    elemento"))
("cero" 1 "dos" 3 "cuatro" 5.322 "a" "b" "c" "un elemento")

> (reverse '("cero" 1 "dos" 3 "cuatro" 5.322))
(5.322 "cuatro" 3 "dos" 1 "cero")

> (member "seis" '("cero" 1 "dos" 3 "cuatro" 5.322))
#f

> (if (member "cero" '("cero" 1 "dos" 3 "cuatro" 5.322))
      "sí está" "no está")
"sí está"

> (list? empty)
#t

> (list? 4)
#f

> (list? '("hola"))
#t
```

9.2. Iteración automática

En Racket no hay ciclos `for` o `while`[1], por lo que se utilizan ciertas funciones predefinidas, propias de los lenguajes funcionales, para recorrer y procesar secuencias (listas) de elementos.

9.2.1. `map`

La primera de ellas, es la función `map` que utiliza los resultados de aplicar una función sobre los elementos de una lista, para generar otra lista. Por ejemplo:

```
> (map sqrt (list 1 2 4 9 16))
(1 1.4142135623730951 2 3 4)

> (map (lambda (x) (+ 1 (sqr x))) (list -5 -4 -3 -2 -1 0 1 2 3 4 5))
(26 17 10 5 2 1 2 5 10 17 26)

> (map (lambda (i) (string-append "¡" i "!"))
       (list "buenos días" "buenas noches"))
("¡buenos días!" "¡buenas noches!")
```

9.2.2. `andmap` y `ormap`

Otras funciones útiles para hacer validaciones de listas son `andmap` y `ormap`. En sus formas más simples, ambas toman como parámetros una función y una lista. En el caso de la primera, retorna `#t` si el resultado de evaluar la función sobre cada elemento de la lista es `#t`; y devuelve `#f` si el resultado de evaluar alguno de los elementos de la lista es `#f`.

La función `ormap` se comporta como se espera, pero aplicando disyunción lógica en lugar de conjunción, que es lo que aplica `andmap`. `ormap` devuelve `#t` si la función se evalúa a verdadero para alguno de los elementos de la lista.

Ejemplos:

```
> (andmap string? '("una cadena" "otra cadena"))
#t

> (andmap string? '("una cadena" "otra cadena" 123456))
#f

> (andmap number? (list 1 3.35 1+8i))
#t

> (andmap number? (list 1 3.35 1+8i "el de la izquierda es un complejo"))
```

[1] En realidad sí hay, puesto que es un lenguaje funcional híbrido. Pero su necesidad es ciertamente algo que está fuera del paradigma funcional.

```
#f

> (ormap (lambda (x) (and (real? x) (positive? x)))
         (list "Sólo complejos:" -1+1i 0+8i (sqrt -4) -9-5i))
#f

> ;;;;;;;; Ejemplo de validación de parámetros con andmap: ;;;;;;;;;;;;;

> (define (suma-tres-enteros-positivos a b c)
    (if (andmap (lambda (x) (and (integer? x) (positive? x)))
                (list a b c))
        (+ a b c)
        "Los parámetros no son enteros positivos"))

> (suma-tres-enteros-positivos 2 3 5)
10

> (suma-tres-enteros-positivos 2 3 -5)
"Los parámetros no son enteros positivos"
```

9.2.3. `filter`

La función `filter` sirve para filtrar elementos de una lista, según el criterio especificado por una función de validación.

Ejemplo:

```
> (filter string? (list 3 "a" "b" 4 5 6))
("a" "b")

> (filter complex? (list "Sólo complejos:" -1+1i 0+8i (sqrt -4) -9-5i))
(-1+1i 0+8i 0+2i -9-5i)

> ; Dejar sólo los elementos que sean impares y múltiplos de
    3;;;;;;;;;;;;;;;;;;;;
> (filter (lambda (x)
            (and (odd? x) ;;impar
                 (= 0 (remainder x 3)))) ;;residuo
          (list 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15))
(3 9 15)

> ;Ahora como una función que recibe una lista como parámetro
    ;;;;;;;;;;;;;;;;;;;;
> (define filtra-impares-y-múltiplos-de-3
    (lambda (lista-números)
      (if (and (list? lista-números)
               (andmap integer? lista-números))
          (filter (lambda (x)
                    (and (odd? x)
                         (= 0 (remainder x 3))))
```

```
                    lista-números)
           "Esta función espera una lista de números"
           )))

> (filtra-impares-y-múltiplos-de-3 (list 1 2 3 4 5 6 7 8 9 10 11 12 13 14
    15))
(3 9 15)

> (filtra-impares-y-múltiplos-de-3 (list "otra cosa"))
"Esta función espera una lista de números"

> (filtra-impares-y-múltiplos-de-3 "otra cosa")
"Esta función espera una lista de números"
```

9.2.4. `for-each`

Existe la necesidad, eventualmente, de recorrer una lista, pero sin considerar el posible resultado de las evaluaciones. Generalmente, este sucede cuando necesitamos mostrar en pantalla cierta información, resultado de procesar una lista. Entonces, puede utilizarse la función nativa `for-each`:

```
> (for-each (lambda (x)
              (display x))
            (list 1 2 3 4 5))
12345
> (for-each (lambda (x)
              (display x)
              (newline))
            (list 1 2 3 4 5))
1
2
3
4
5

> ;; Compare los resultados entre map y for-each: ;;;;;;;;;;;;;;;;

> (for-each integer? (list 2 3.1 4 5 6.6))
> (map      integer? (list 2 3.1 4 5 6.6))
(#t #f #t #t #f)
```

La función `for-each`, a diferencia de `map`, ignora el resultado de las evaluaciones de la función sobre los elementos de la lista. Con `for-each` sólo importan los efectos colaterales de las invocaciones (como las escrituras en pantalla o en archivo), no su resultado.

9.2.5. Versiones generales de las funciones de iteración

Las funciones `map`, `for-each`, `andmap` y `ormap` pueden manipular múltiples listas, en lugar de sólo una. Las listas deben tener la misma longitud, y la función dada debe aceptar un parámetro por cada lista:

```
> (map + (list 1 2 3 4 5) (list 10 100 1000 10000 100000))
(11 102 1003 10004 100005)

> (map (lambda (s n) (substring s 0 n))
       (list "agua loca" "hoja de papel" "dulcera")
       (list 4 4 7))
("agua" "hoja" "dulcera")

> ;;;;Compare otra vez el comportamiento de map vs. for-each:
    ;;;;;;;;;;;;;;;
> (map / (list 1 2 3 4 5) (list 5 4 3 2 1))
(1/5 1/2 1 2 5)

> (for-each (lambda (a b)
              (printf "~a\n" (/ a b)))
            (list 1 2 3 4 5) (list 5 4 3 2 1))
1/5
1/2
1
2
5
```

9.3. Iteración manual

Eventualmente es necesario procesar listas a más bajo nivel que el que proveen funciones como `map`. En esos casos, se requiere de mecanismos *más primitivos* como los siguientes:

- `first` devuelve el primer elemento de una lista no vacía
- `rest` devuelve una lista con los elementos de una lista no vacía, sin su primer elemento (el resultado puede ser una lista vacía si la lista de entrada tenía sólo un elemento)
- `cons` concatena un elemento a una lista, produciendo una lista nueva
- `cons?` verifica si un elemento es una lista no vacía (lo contrario de `empty?` y de `null?`)

Ejemplos:

```
> (first (list 1 2 3))
1

> (rest (list 1 2 3))
(2 3)

```

```
> (cons "cabeza" empty)
("cabeza")

> (cons "nueva" (cons "cabeza" empty))
("nueva" "cabeza")

> (empty? empty)
#t

> (empty? (cons "cabeza" empty))
#f

> (cons? empty)
#f

> (cons? (cons "cabeza" empty))
#t
```

9.3.1. Aplicación

Teniendo a nuestra disposición estas funciones de *bajo nivel* para manipular funciones, podríamos construir nuestras propias funciones de longitud de lista y de mapeo de lista:

```
(define (longitud L)
  (cond
    [(empty? L) 0]
    [else (+ 1 (longitud (rest L)))]))

(define (mapear f L)
  (cond
    [(empty? L) empty]
    [else (cons (f (first L))
                (mapear f (rest L)))]))
```

También podemos hacer funciones que procesen listas de manera básica. Por ejemplo considere la siguiente función para generar listas de números enteros:

```
(define secuencia-de-enteros
  (lambda (num-elementos inicio paso)
    (define (aux i contador lista)
      (if (>= contador num-elementos)
          (reverse lista)
          (aux (+ i paso) (add1 contador) (cons i lista))
          ))
    (if (and (exact-nonnegative-integer? num-elementos)
             (integer? inicio)
             (integer? paso))
        (aux inicio 0 empty)
        (error "Error en los parámetros")
        )
```

```
14      ))
```

9.4. Pares y listas

La función `cons` acepta dos parámetros, y el segundo no necesariamente debe ser una lista. En el caso que como segundo argumento se le pase algo que no sea una lista, la función cons devuelve un **Par** o **Pareja**.

Un Par en Racket no es otra cosa que dos elementos (de cualquier tipo), ligados entre sí. Una lista no vacía, de hecho, es un par compuesto por un elemento (el primero) y una lista (que puede ser vacía).

La notación que utiliza Racket para representar los pares es la de los elementos, encerrados entre paréntesis y separados por un espacio en blanco, un punto y otro espacio en blanco:

```
1  > (cons 1 2)
2  (1 . 2)
3
4  > (cons "una cadena" 4)
5  ("una cadena" . 4)
```

Hay una función equivalente a `cons?` con más sentido para los pares: `pair?`. También hay funciones correspondientes a `first` y `rest` para pares: `car` y `cdr`. Estas últimas funcionan con cualquier tipo par (incluyendo las listas no vacías) y las primeras, sólo funcionan con listas no vacías, pero no con pares.

Ejemplos:

```
1  > (define vacío '())
2  > (define par (cons 1 2))
3  > (define lista (cons 1 (cons 2 '())))
4
5  > (pair? vacío)
6  #f
7  > (pair? par)
8  #t
9  > (pair? lista)
10 #t
11 > (car par)
12 1
13 > (car lista)
14 1
15 > (cdr par)
16 2
17 > (cdr lista)
18 (2)
19 > (list? vacío)
20 #t
```

```
> (list? par)
#f
> (list? lista)
#t
```

9.4.1. Convención de impresión

Racket tiene una convención para imprimir los pares, que puede llegar a ser muy confusa. Por ejemplo, considere el siguiente resultado:

```
> (cons 0 (cons 1 2))
(0 1 . 2)
```

Lo anterior es un par, cuyo segundo elemento es otro par que no es una lista.

La regla para la impresión es la siguiente:

> Usar siempre la notación de punto, pero si el punto está inmediatamente seguido de una apertura de paréntesis, entonces, remover el punto, el paréntesis de apertura y el correspondiente paréntesis de cierre.

Así, `(0 . (1 . 2))` se convierte en `(0 1 . 2)`. La utilidad de esta, aparentemente, extraña regla, es para volver legibles las listas, ya que, por ejemplo, `(1 . (2 . (3 . ())))` –que es una lista de tres elementos en su notación de punto– se convierte en `(1 2 3)`, lo cual es más fácil de leer.

9.4.2. Notación infija

Existe una convención particular en Racket que, aunque no es tradicional en Lisp y otros dialectos de Scheme, puede mejorar la legibilidad de ciertas partes de nuestras funciones.

> Un par de *puntos* pueden aparecer alrededor de un solo elemento en una secuencia parentizada, mientras el elemento no sea ni el primero ni el último. Esta convención de sintaxis ejecuta una conversión que mueve el elemento entre los *puntos* hacia el frente de la secuencia parentizada.

Esta convención posibilita una especie de notación infija, a la cual estamos más acostumbrados:

```
> (1 . + . 2 3 4 5)
15

> '(1 . + . 2 3 4 5)
(+ 1 2 3 4 5)

> (1 . < . 2)
#t
```

```
9
10  > '(1 . < . 2)
11  (< 1 2)
12
13  > (1 2 3 . * . 4)
14  24
15
16  > '(1 2 3 . * . 4)
17  (* 1 2 3 4)
```

Ejercicios de Listas e iteración

1. Dada una lista desordenada de números enteros ordenar dicha lista de mayor numero a menor, tomar en cuenta que pueden existir números repetidos en dicha lista (No utilizar la función `sort`).
2. Dada una lista compuesta por números enteros eliminar los elementos repetidos de dicha lista. Retornar una nueva lista. Nota: Se pide *retornar* un lista, es decir que no se pide usar `display`, por lo demás, se puede utilizar cualquier primitiva.
3. Dada una lista compuesta por cadenas, construir una nueva lista a partir de la anterior formada por los elementos que no estén repetidos. Ejemplo para mayor comprensión: `'("hola" "mundo" "mundo")` la nueva lista será `'("hola")`.
4. Dada una lista compuesta por cadenas, construir una cadena formada por cada cadena que se encuentre en la lista. Ejemplo `'("hola" "mundo")` retornará `"holamundo"`. (Note que de ser la segunda cadena `"  mundo"` se retornaría `"hola mundo"`. Puede usar las primitivas que desee. Se recomienda leer sobre `string-append`.
5. Dada una lista compuesta por listas, retornar verdadero si todas las sublistas están vacías y falso si por lo menos una posee algún elemento. Puede usar cualquier primitiva.
6. Dada una lista compuesta por 5 números no repetidos, retornar el número mayor de dicha lista (ojo se pide retornar no usar `display`).
7. Dada una lista compuesta por 5 cadenas no repetidas, retornar la cadena de mayor longitud. En caso de que existan 2 o más cadenas de igual longitud y estas resulten las de mayor longitud, retornar ambas cadenas (para facilitar el ejercicio puede retornar la cadena o las cadenas dentro de una lista).
8. Dada una lista compuesta por números enteros, desplegar la cantidad de números pares y la cantidad de números impares.
9. Dada una lista compuesta por números enteros, retornar la sumatoria de todos los números pares.
10. Dada una lista compuesta por números enteros y dado un número, retornar `#t` si el número se encuentra en la lista y `#f` si dicho número no se encuentra (NO USAR `member`).
11. Dada una lista compuesta por números y dado un número, eliminar dicho número de la lista si este se encuentra en ella (puede usar cualquier primitiva).

12. Dada una lista compuesta por tres puntos (un punto es una lista, ejemplo `'(1 2)` es un punto $x \to 1$, $y \to 2$) retornar `#t` si dichos puntos forman un triángulo equilátero, y `#f` en caso contrario (es equilátero si sus tres lados son iguales). NOTA : Fórmula de distancia entre los puntos (x_1, y_1) y (x_2, y_2): $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
13. Dada una lista compuesta por listas, retornar una lista compuesta por los elementos de cada sublista. Ejemplo `'( (1 2) (2 3))` retornará `(1 2 2 3)`. Como puede observar, pueden existir elementos repetidos.
14. Dada una lista compuesta por cadenas, retornar la cantidad de vocales dentro de dicha lista. Ejemplo `'("hola" "mundo")` retornará 4.
15. Dada una lista compuesta por cadenas, retornar la lista compuesta por las cadenas sin sus vocales. Ejemplo: `'("hola" "mundo")` retornará `'("hl" "mnd")` note que se eliminaron las vocales. El orden de las cadenas no debe cambiar.
16. Dada una cadena, pasar cada letra de la cadena a una lista, ejemplo `"hola"` se convierte en `'("h" "o" "l" "a")`. Note que no se piden los caracteres si no las letras en forma de cadena.
17. Dada una lista compuesta por cadenas, ordenar dicha lista tomando como criterio la longitud de las cadenas (No usar `sort`).
18. Elaborar una función que reciba como parámetro una lista de números enteros y positivos, la función evaluará la lista de números, si la lista está ordenada de mayor a menor, la función retornará dos listas (usando la función `values`) la primer lista contendrá los números pares de la lista original, respetando el mismo orden de mayor a menor, y la segunda lista contendrá los números impares de la lista original respetando el orden de la lista original; en caso contrario, es decir si la lista pasada de parámetro está desordenada, se retornará la lista ordenada de mayor a menor.

10 Recursión

10.1. Recursión por Posposición de trabajo

En el caso del siguiente código de la función `longitud`, el tipo de recursión usada es **posposición de trabajo**:

```
(define (longitud L)
  (cond
    [(empty? L) 0]
    [else (+ 1 (longitud (rest L)))]))
```

Y al evaluar, por ejemplo, `(longitud (list "a" "b" "c"))` se da este proceso:

```
-> (longitud (list "a" "b" "c"))
 = (+ 1 (longitud (list "b" "c")))
 = (+ 1 (+ 1 (longitud (list "c"))))
 = (+ 1 (+ 1 (+ 1 (longitud (list)))))
 = (+ 1 (+ 1 (+ 1 0)))
 = (+ 1 (+ 1 1))
 = (+ 1 2)
 = 3
```

Como puede verse, se tienen que apilar todos los cálculos y todas las sumas quedan *pospuestas* hasta que se alcanza el caso trivial de la recursión, que en este caso es cuando se encuentra una lista vacía. Si la longitud de la lista es demasiado grande, provocará un gran consumo de memoria.

Esto no es algo "extraño", sin embargo resulta ser ineficiente en Racket, ya que este lenguaje provee una optimización importante para la recursión de cola, que se explica a continuación.

10.2. Recursión de Cola

Considere la siguiente versión de `longitud` con recursión de cola:

```
(define (longitud L)
  ; función local longitud-aux:
  (define (longitud-aux L longitud-actual)
    (cond
      [(empty? L) longitud-actual]
```

```
6         [else (longitud-aux (rest L) (+ longitud-actual 1))]))
7     ; este es el cuerpo de longitud, que llama a longitud-aux:
8     (longitud-aux L 0))
```

Ahora veamos el cálculo de `(longitud (list “a” “b” “c”))`:

```
1  -> (longitud (list "a" "b" "c"))
2   = (longitud-aux (list "a" "b" "c") 0)
3   = (longitud-aux (list "b" "c") 1)
4   = (longitud-aux (list "c") 2)
5   = (longitud-aux (list ) 3)
6   = 3
```

Note que no hay retornos pendientes en ningún momento, tampoco hay cálculos (en este caso, sumas) que queden pendientes en cada paso de la recursión.

En Racket, cuando una función se reduce a una expresión cuyos parámetros son totalmente conocidos, toda la memoria de la función es liberada y ya no queda rastro de su invocación. Esto no sólo sucede con la recursión de cola, sino con cualquier llamada para la cual no queden cálculos pendientes.

Esta es una diferencia importante de Racket con respecto a otros lenguajes de programación no funcionales, ya que en otros lenguajes, aún haciendo recursión de cola, siempre queda memoria de las llamadas anteriores, apiladas esperando algún `return`, `end` o equivalente. Esto provoca que la cantidad de memoria necesaria para ejecutar el procedimiento recursivo es aproximadamente lineal a la profundidad de la llamada. En Racket, la recursión de cola se ejecuta en una cantidad de memoria fija, para toda la ejecución de la función recursiva.

Queda entonces, la atenta invitación a utilizar recursión de cola en los programas hechos con Racket, siempre que sea posible.

11 Tipos de dato integrados del lenguaje

Aquí se describen los principales tipos integrados, nativos de Racket. El lenguaje incluye muchos otros tipos de datos complejos que no serán abordados aquí.

11.1. Booleanos

El tipo más simple de Racket es el **booleano** o **lógico**. Sólo tiene dos valores constantes, que son `#t` para verdadero y `#f` para falso (también se aceptan las formas `#F` y `#T`, pero las versiones en minúsculas son preferidas).

Existe la función `boolean?` que verifica si un valor es una de las dos constantes lógicas, `#t` o `#f`:

```
> (boolean? 0)
#f
> (boolean? #f)
#t
```

A pesar de que se espera un valor de verdad en las expresiones de prueba de las construcciones `if`, `cond`, `and`, `or` y otras, todos los valores posibles en Racket, excepto `#f` se evalúan como verdadero:

```
> (define (mostrar-valor-de-verdad v) (if v #t #f))

> (mostrar-valor-de-verdad "")
#t

> (mostrar-valor-de-verdad "no")
#t

> (mostrar-valor-de-verdad empty )
#t

> (mostrar-valor-de-verdad '(1 2 3) )
#t

> (mostrar-valor-de-verdad #() )
#t

> (mostrar-valor-de-verdad #(1 2 3) )
```

```
#t

> (mostrar-valor-de-verdad #\a )
#t
```

11.2. Números

A continuación se presenta el tratamiento de los **números** en Racket.

Un valor numérico se puede validar con la función `number?`:

```
> (number? 3)
#t

> (number? 3.1416)
#t

> (number? "3")
#f

> (number? 5+8i)
#t

> (number? 5/8)
#t

> (number? 3.45e-200)
#t
```

11.2.1. Clasificación

Hay dos formas de clasificar números en Racket: Por exactitud y por conjuntos.

Clasificación por Exactitud

En Racket, un *número* es **exacto** o **inexacto**.

Los *números exactos* son:

1. Los enteros
2. Los racionales
3. Los complejos con parte real exacta y parte imaginaria exacta

Los *números inexactos* son:

1. Los reales de coma flotante
2. Los complejos con parte real inexacta o parte imaginaria inexacta

Existen las funciones `exact?` e `inexact?` para determinar si un número pertenece a uno de los dos tipos anteriores.

```
> (exact? 7)
#t

> (inexact? 7)
#f

> (inexact? empty)
. . inexact?: expects argument of type <number>; given ()

> (inexact? "")
. . inexact?: expects argument of type <number>; given ""

> (inexact? 8.999993-8.325421i)
#t

> (inexact? 7/8)
#f
```

Pueden también utilizarse las funciones `exact->inexact` e `inexact->exact` para convertir de un tipo a otro:

```
> (exact->inexact 1/3)
0.3333333333333333

> (exact->inexact (/ 7 8))
0.875

> (inexact->exact 0.33333333333333333333333333333333)
6004799503160661/18014398509481984
```

Además, existe una forma de forzar la representación, como exacto o inexacto, de un número, independientemente de la forma en que se escriba. Con los prefijos `#e` y `#i`:

```
> #e0.2
1/5

> #i1/5
0.2

> #i4+5i
4.0+5.0i
```

Propagación de la exactitud Con los operadores aritméticos básicos, los números exactos se mantienen exactos tras los cálculos y los inexactos se mantienen inexactos a través de los cálculos:

```
> (define (sigma f a b)
    (if (= a b)
        0
        (+ (f a) (sigma f (+ a 1) b))))

> (sigma (lambda (x) (/ 1   x)) 5 8)
107/210

> (sigma (lambda (x) (/ 1.0 x)) 5 8)
0.5095238095238095
```

Clasificación por Conjuntos

Tal como en la matemática tradicional, los números se categorizan por la jerarquía del conjunto al que pertenecen: $\mathbb{Z} \subset \mathbb{Q} \subset \mathbb{R} \subset \mathbb{C}$ (es decir, los enteros están incluídos en los racionales, estos en los reales, y estos en los complejos):

```
> (integer? -5)
#t

> (rational? -5/9)
#t

> (real? -5/9)
#t

> (complex? -5/9)
#t
```

11.2.2. Otras bases

La base para todos los números (desde los enteros hasta los complejos) es **10**, pero puede forzarse a que sea base 2, base 8 o base 16 con los prefijos `#b`, `#o`, `#x`, respectivamente:

```
> #b11
3

> #o10
8

> #xff
255

> #b111.01
7.25

> #xf/5
3
```

11.2.3. Comparaciones

Los números exactos pueden ser comparados con la función `=` o con `equal?`, pero los números inexactos, debido a su propia naturaleza, deberían ser comparados *por proximidad* en lugar de *por igualdad*, ya que su representación no es exacta:

```
> (= 3 6/2)
#t

> (= 4+8i 8/2+24/3i)
#t

> (equal? 4+8i 8/2+24/3i)
#t

> (= 4.0+8.0i 8/2+24/3i)
#t

> (= 4.0 4)
#t

> (= 0.1 1/10)
#f

> (inexact->exact 0.1)
3602879701896397/36028797018963968

> (let ([a 0.1]
        [b 1/10]
        [tolerancia 0.00001])
    (< (abs (- a b)) tolerancia))
#t
> (define (¿son-reales-iguales? a b tol)
          (< (abs (- a b)) tol)
          )
```

11.2.4. Constantes especiales

Existen cuatro constantes especiales, definidas por la IEEE:

- `+inf.0`/`-inf.0` que resultan de sobrepasar la capacidad de representación de los números en coma flotante, por arriba o por abajo, respectivamente.
- `+nan.0`/`-nan.0` que resultan de cálculos indeterminados como cero entre cero, infinito entre infinito, cero por infinito, infinito menos infinito, etc.

```
> (/ 8.0 0.0)
+inf.0

> -5.38e700
```

```
-inf.0

> 1.79e-400
0.0

> 1.79e308
1.79e+308

> 1.79e309
+inf.0

> ;; 'NaN' significa: Not a Number.

> (/ 0.0 0.0)
+nan.0

> (/ +inf.0 -inf.0)
+nan.0

> (* 0.0 -inf.0)
+nan.0

> (+ +inf.0 -inf.0)
+nan.0
```

Si estos valores especiales se pasan como parámetro a alguna función que espere números, su resultado será del mismo tipo (excepto para algunas funciones para las que tiene significado):

```
> (cos (* (+ (* 0.0 -inf.0) 1) 9))
+nan.0

> (atan +inf.0)
1.5707963267948966

> (* 2 (atan +inf.0)) ;pi
3.141592653589793
```

11.3. Caracteres

Un **caracter** en Racket, es un valor escalar Unicode (igual que en otros lenguajes de programación como *Java*).

Los caracteres literales, se expresan como una secuencia `#\` seguido del caracter correspondiente, si es que estos tienen una representación imprimible y escribible:

```
> #\0
#\0
> #\a
#\a
```

```
> #\newline
#\newline
> #\space
#\space
> #\&
#\&
```

A pesar que un caracter se corresponda con un entero en Racket, a diferencia de otros lenguajes (como *Java* o *C*), no se pueden mezclar directamente con los números. Para poder hacerlo, se utilizan las funciones `char->integer` e `integer->char`:

```
> (integer->char 32)
#\space
> (integer->char 65)
#\A
> (integer->char 92)
#\\
> (char->integer #\A)
65
> (char->integer #\a)
97
> (char->integer #\ü)
252
> (char->integer #\Ü)
220
> (char->integer #\ĝ)
285
> (char->integer #\δ)
948
> (char->integer #\ओ)
2708
> (char->integer #\香)
12217
> (integer->char 12218)
#\馬
>
```

Si algún caracter no tiene una representación imprimible, este siempre se puede mostrar con la notación Unicode tradicional de una letra `u` minúscula y un número hexadecimal de dos bytes:

```
> (integer->char 17)
#\u0011
> (char->integer #\u011D)
285
```

Existen ciertas funciones útiles para manipular y procesar caracteres:

```
> (char-alphabetic? #\ĝ) ;; Letra 'ĝo' del Esperanto
#t
> (char-alphabetic? #\u2fba) ;;Caracter chino
#f
> #\u2fba
#\馬
> (char-alphabetic? #\ℵ) ;;Letra 'aleph' del hebreo
#t
>

(char-numeric? #\8)
#t
> (char-numeric? #\٣) ;Número 3 en árabe
#t
>

(char-whitespace? #\newline)
#t
> (char-whitespace? #\tab)
#t
>

(char-downcase #\E)
#\e
> (char-upcase #\ϕ) ;;Letra 'ef' cirílica minúscula
#\Φ
> (char-upcase #\ĝ)
#\Ĝ
> (char-downcase #\Δ) ;;Letra 'delta' mayúscula
#\δ
>

(char? #\5)
#t
> (char? "a")
#f
> (char=? #\u #\ü) ;;Comparación entre caracteres, también se puede con 'equal?'
.
#f
> (char-ci=? #\Ä #\ä) ;;Comparación entre caracteres, sin distinción de caso (Ig
nore Case).
#t
> 
```

11.4. Cadenas

Una **cadena** es un arreglo de caracteres de longitud fija. Como en muchos otros lenguajes, se escriben entre comillas dobles.

Como en otros lenguajes, para poder escribir comillas dobles dentro de la cadena, hay que utilizar la secuencia \". Esto se conoce como secuencia de escape. De la misma manera, hay varias secuencias de escape, como \\ para escribir una pleca, `\n` para una nueva línea, `\r` para un retorno de carro. Y para escribir un caracter dado su código octal, \777 y `\uFFFF` para escribirlo en función de su código hexadecimal Unicode.

La función `display` escribe los caracteres de la cadena, pero sin las comillas, a diferencia de lo que sucede cuando el resultado de una expresión es una cadena.

Ejemplos:

```
> "Cadena"
"Cadena"
> "йа алиа руслингжа фразо"
"йа алиа руслингжа фразо"
> "eñoŝanĝo ĉiuĵaŭde"
"eñoŝanĝo ĉiuĵaŭde"
> "\u03bb es el símbolo del cálculo lambda"
"λ es el símbolo del cálculo lambda"
> (display "ĊăḊĒṇą\n")
ĊăḊĒṇą
> (display "una cadena con \"comillas\" en medio\n")
una cadena con "comillas" en medio
> (display "una cadena\ncon dos líneas\n")
una cadena
con dos líneas
> "otra cadena\ncon dos líneas y una \\"
"otra cadena\ncon dos líneas y una \\"
>
```

Hay tres funciones básicas para la creación y manipulación de cadenas:

- `string` forma una nueva cadena a partir de una serie de caracteres;
- `string-ref` devuelve un caracter de una cadena, dada su posición; y
- `string-length` devuelve su longitud medida en caracteres

Ejemplos:

```
1  > (string #\H #\o #\l #\a)
2  "Hola"
3
4  > (string)
5  ""
6  > (string-ref "Hola" 0)
7  #\H
8
9  > (string-ref "Hola" 3)
10 #\a
11
12 > (string-length "Hola")
13 4
```

11.4.1. Cadenas mutables

Por defecto, los literales de cadena escritos en el código fuente, se convierten en cadenas **inmutables**, es decir, que no pueden ser cambiados durante el curso de su existencia como objetos del programa. Pero si requerimos alterar una cadena durante la ejecución, debemos crear una **cadena mutable**. Veamos las funciones para crear una cadena mutable y alterar su contenido:

- `make-string` recibe una longitud para la nueva cadena mutable y opcionalmente un caracter de relleno, por defecto el caracter nulo (`\u0000`).

- `string-set!` modifica un caracter de una cadena mutable, dada su posición.
- `string->immutable-string` convierte una cadena mutable en su versión inmutable (si recibe una inmutable, la devuelve a ella misma).
- `immutable?` verifica si un objeto es inmutable –no sólo las cadenas pueden ser mutables– .
- `string-copy!` copia total o parcialmente el contenido de una cadena –mutable o inmutable– a otra cadena mutable.

```
> (make-string 4 #\c)
"cccc"

> (define cadena-mutable (make-string 4 #\c))

> (string-length cadena-mutable)
4

> (string-ref cadena-mutable 2)
#\c

> (string-set! cadena-mutable 2 #\a)
> cadena-mutable
"ccac"

> (define cadena-inmutable (string->immutable-string cadena-mutable))

> (immutable? cadena-inmutable)
#t

> (immutable? cadena-mutable)
#f

> (define otra-cadena-mutable (make-string 10))

> otra-cadena-mutable
"\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000\u0000"

> (string-copy! otra-cadena-mutable 0 "buen día")

> otra-cadena-mutable
"buen día\u0000\u0000"
```

11.4.2. Comparación entre cadenas

La comparación entre cadenas se realiza con las siguientes funciones:

- `string=?`, `string<?`, `string<=?`, `string>?`, `string>=?` para hacer comparaciones simples en función del orden relativo de los caracteres en el estándar Unicode.

- `string-ci=?`, `string-ci<?`, `string-ci<=?`, `string-ci>?`, `string-ci>=?` para hacer comparaciones insensibles al caso (sin distinción entre mayúsculas o minúsculas).
- `string-locale=?`, `string-locale<?`, `string-locale>?`, `string-locale-ci=?`, `string-locale-ci<?`, `string-locale-ci>?` para hacer comparaciones en función de ciertas consideraciones alfabéticas y lexicográficas, en lugar de sólo por las posiciones en Unicode.

Ejemplos:

```
> (string-ci<? "algo" "Básico")
#t

> (string<? "algo" "Básico")
#f

> (string-locale>? "árbol" "burro")
#f

> (string>? "árbol" "burro")
#t
```

11.4.3. Otras funciones de cadena

- `string-append` devuelve una nueva cadena mutable, resultado de concantenar una serie de cadenas.
- `string->list` devuelve una lista de todos los caracteres correspondientes a una cadena.
- `list->string` devuelve una nueva cadena mutable que contiene todos los caracteres de la cadena proporcionada.
- `substring` devuelve una nueva cadena mutable que es un subconjunto de la cadena proporcionada.

Ejemplos:

```
> (string-append "Esta " "es una cadena" " unida" (make-string 3 #\-))
"Esta es una cadena unida---"

> (string->list "cadena")
(#\c #\a #\d #\e #\n #\a)

> (list->string '(#\C #\a #\d #\e #\n #\a))
"Cadena"

> (substring "0123456789" 1 7)
"123456"

> (substring "0123456789" 4)
```

```
"456789"

> (substring "0123456789" 4 5)
"4"
```

11.5. Bytes y Cadenas de Bytes

Un **Byte**, en Racket, es un entero exacto en el intervalo cerrado `[0,255]` (o en hexadecimal, `[#x0,#xff]`). La función `byte?` reconoce este tipo de números. No es en realidad un tipo específico, sino un subconjunto de los números enteros.

Ejemplos:

```
> (byte? #xfa)
#t
> (byte? 56)
#t
> (byte? 256)
#f
```

Su utilidad radica en que sirven para construir **cadenas de bytes**, que se utilizan para comunicaciones de bajo nivel. Estas cadenas no tienen codificación Unicode, sino ASCII. En modo interactivo, se muestran como cadenas normales precedidas de un `#`. Y al igual que con las cadenas normales –las cadenas Unicode– son por defecto *inmutables*. Cuando se imprimen en pantalla, se usa la codificación ASCII, y si un byte no es imprimible, se muestra su valor en octal.

Estas son algunas funciones para manipular cadenas de bytes:

- `bytes-ref` devuelve un byte de la cadena de bytes dada su posición.
- `make-bytes` devuelve una nueva cadena de bytes *mutable*, dada su longitud y un byte de relleno.
- `bytes-set!` cambia un byte de una cadena de bytes mutable, dada su posición y el nuevo byte.
- `bytes?` verifica si un valor es una cadena de bytes.

Ejemplos:

```
> #"aBcD"
#"aBcD"
> (define cad-bytes #"aBcD")
> (bytes-ref cad-bytes 0) ;;La letra 'a'
97
>
> (define otra-cadena (make-bytes 4 97))
> otra-cadena
```

```
#"aaaa"
> (bytes-set! otra-cadena 0 98) ;;La letra 'b'
> otra-cadena
#"baaa"
>
> (bytes-set! otra-cadena 3 0)
> (bytes-set! otra-cadena 2 10)
> (bytes-set! otra-cadena 1 5)
> otra-cadena
#"b\5\n\0"
```

11.6. Símbolos

Un **símbolo** es como una cadena inmutable, pero sin la posibilidad de acceder a sus caracteres. Su utilidad radica en que son buenos para servir como etiquetas –o valores constantes–, o enumeraciones para las funciones.

Hay algunas funciones que sirven para manipularlos:

- `symbol?` para verificar si un valor es símbolo o no.
- `string->symbol` convierte una cadena en su correspondiente símbolo.
- `symbol->string` convierte un símbolo en su respectiva cadena.

Un símbolo se imprime como un identificador, pero puede estar compuesto por caracteres no permitidos en los identificadores —espacios en blanco y `( ) [ ] { } " , ' ` ; # | \`—, en cuyo caso, se imprime como una secuencia de caracteres, encerrados en barras verticales: `||`.

```
> 'símbolo
símbolo

> (symbol? 'símbolo)
#t

> (symbol? "símbolo")
#f

> (string->symbol "este-es-un-símbolo")
este-es-un-símbolo

> (string->symbol "este es un símbolo")
|este es un símbolo|
```

Por ejemplo, considere el siguiente archivo de código:

```
#lang racket
;símbolo.rkt
```

```
3
4  (define secuencia-de-enteros
5    ;; significado-fin puede ser 'número-de-elementos 'valor-final
6    (lambda (inicio fin paso significado-fin)
7      (define (aux-num-elementos i contador lista)
8        (if (>= contador fin)
9            (reverse lista)
10           (aux-num-elementos (+ i paso) (add1 contador) (cons i lista))
11           ))
12     (define (aux-valor-final i lista)
13       (if (>= i fin)
14           (reverse lista)
15           (aux-valor-final (+ i paso) (cons i lista))
16           )
17       )
18     (if (and (integer? fin)
19              (integer? inicio)
20              (integer? paso))
21         (if (equal? significado-fin 'número-de-elementos)
22             (if (exact-nonnegative-integer? fin)
23                 (aux-num-elementos inicio 0 empty)
24                 (error "El número de elementos debe ser no negativo")
25                 )
26             (if (equal? significado-fin 'valor-final)
27                 (aux-valor-final inicio empty)
28                 (error "El último parámetro se esperaba como 'número-de-
                       elementos o como 'valor-final")
29                 )
30             )
31         (error "Error en los parámetros. Los primeros tres deben ser
                enteros.")
32         )
33     ))
```

Tiene la siguiente salida:

```
1  > (secuencia-de-enteros 2 5 2 'número-de-elementos)
2  (2 4 6 8 10)
3
4  > (secuencia-de-enteros 2 5 2 'valor-final)
5  (2 4)
6
7  > (secuencia-de-enteros 0 10 3 'número-de-elementos)
8  (0 3 6 9 12 15 18 21 24 27)
9
10 > (secuencia-de-enteros 0 10 3 'valor-final)
11 (0 3 6 9)
```

11.7. Palabras clave

Las **palabras clave** son elementos de la forma `#:palabra`. No constituyen una expresión en sí mismas y sirven para *el paso de parámetros por nombre*. Su utilidad se explica en la subsección 12.2.4 en la página 92.

11.8. Pares y listas

Los **pares y listas** son tratados en el capítulo 9, en sus formas *inmutables*. Pero también hay pares y listas *mutables*, de las que sí se hablará aquí.

Hay dos detalles importantes sobre la *mutabilidad* de los pares:

1. La lista vacía no es mutable ni inmutable.
2. Las funciones `pair?` y `list?` sólo reconocen pares y listas inmutables.

A continuación, veamos cómo manipular **pares (y listas) mutables**:

- `mcons` construye un par mutable (puede ser una lista mutable).
- `mpair?` verifica si un par es mutable.
- `set-mcar!` para cambiar el primer elemento de un par mutable.
- `set-mcdr!` para cambiar el segundo elemento de un par mutable.
- `mcar` y `mcdr` devuelven el primer y segundo elemento de un par mutable respectivamente.

Ejemplos:

```
> (mpair? (mcons 3 '()))
#t

> (mcons 3 '())
{3}

> (mcons 3 (mcons 2 (mcons 1 '())))
{3 2 1}

> (define lista-mutable (mcons 3 (mcons 2 (mcons 1 '()))))

> (define par-mutable (mcons "a" "b"))

> par-mutable
{"a" . "b"}

> (pair? lista-mutable)
#f

```

```
> (pair? par-mutable)
#f

> (mpair? lista-mutable)
#t

> (mpair? par-mutable)
#t

> (mcdr lista-mutable)
{2 1}

> (set-mcar! par-mutable "algo más")

> par-mutable
{"algo más" . "b"}

> lista-mutable
{3 2 1}

> (set-mcdr! lista-mutable "??")

> lista-mutable
{3 . "??"}
```

Los pares y listas mutables se imprimen encerrados en llaves, pero es sólo una convención para imprimir, pero no para escribirlas directamente.

11.9. Vectores

Un **vector** es un arreglo de longitud fija de valores arbitrarios. A diferencia de una lista, que es una lista lineal de nodos enlazados en memoria, un vector soporta acceso a sus elementos –lectura y escritura– en tiempo constante. Esa es básicamente su mayor diferencia.

Otra diferencia, es que al imprimirse, un vector se muestra como una lista precedida por un `#`. Cuando se escribe un vector con esta notación, por defecto es inmutable.

Algunas funciones básicas para manipular vectores son:

- `vector` construye un nuevo vector mutable conteniendo los parámetros de la función.
- `vector?` verifica si su parámetro es un vector.
- `vector-ref` devuelve un elemento de un vector en función de su posición.
- `list->vector` convierte una lista en un vector con los mismos elementos.
- `vector->list` convierte un vector en su representación de lista.
- `vector-set!` modifica un valor de un vector mutable dada su posición.

- `vector-length` devuelve la longitud de un vector.

Ejemplos:

```
> #(1 "dos" 3.1) ;; Esto genera un nuevo vector inmutable
#(1 "dos" 3.1)

> (define v #(1 "dos" 3.1))

> (vector-ref v 0)
1
> (vector-ref v 2)
3.1

> (vector->list v)
(1 "dos" 3.1)

> (list->vector '("a" "b" "c"))
#("a" "b" "c")

> (vector 1 2 3)
#(1 2 3)

> (define v (vector 0 1 2))
> (vector-set! v 1 "uno")
> v
#(0 "uno" 2)
```

También hay algunas otras funciones para manipular vectores –mutables e inmutables– de manera parecida a las listas:

- `make-vector` crea un vector mutable de un tamaño especificado y opcionalmente un valor de relleno.
- `vector-immutable` igual que vector pero devuelve un vector inmutable.
- `vector->immutable-vector` devuelve un vector inmutable dado otro vector (que si ya es inmutable, es el mismo devuelto).
- `vector-copy!` copia total o parcialmente el contenido de un vector (mutable o inmutable) a un vector mutable.

11.10. Tablas Hash

Una **tabla hash** es una estructura de dato que implementa el *mapeo* de *claves*, a *valores* arbitrarios. Tanto las *claves* como los *valores* pueden ser valores arbitrarios en Racket, y el tiempo de acceso a los *valores* suele ser en tiempo constante, a diferencia del tiempo de acceso a los vectores que siempre es constante y a diferencia del de las listas que es linealmente creciente dependiendo de la posición del elemento a accesar.

Tal como con otros tipos de dato en Racket, existe una gran cantidad de funciones nativas para manipular tablas hash, además de existir en versión mutable e inmutable:

- `hash?` verifica si un elemento es una tabla hash.
- `make-hash` no recibe parámetros y devuelve una tabla hash *mutable* vacía.
- `hash-set!` agrega una asociación dentro de una tabla hash mutable, sobreescribiendo cualquier asociación previa para la clave indicada.
- `hash-set` toma una tabla hash inmutable, una clave y un valor, y devuelve otra tabla hash inmutable equivalente a la anterior más la nueva asociación entre la clave y el valor indicados.
- `hash-ref` devuelve el valor al que corresponde una clave indicada, dentro de una tabla hash indicada, si es que existe.
- `hash-remove` elimina una clave y su respectivo valor de una tabla hash mutable.
- `hash-count` devuelve el tamaño de una tabla hash medida en número de pares contenidos en ella.

Ejemplos:

```
> (define ht-lenguajes (make-hash))
> (hash-set! ht-lenguajes "c" '(estructurado bibliotecas compilado))
> (hash-set! ht-lenguajes "java" '(oo paquetes compilado))
> (hash-set! ht-lenguajes "racket" '(funcional módulos interpretado))
> (hash-ref ht-lenguajes "java")
(oo paquetes compilado)
> (hash-ref ht-lenguajes "python")
. . hash-ref: no value found for key: "python"
> (hash-ref ht-lenguajes "python" "no está")
"no está"

> (hash-count ht-lenguajes)
3
> (hash-set! ht-lenguajes "python" '(multiparadigma módulos interpretado))
> (hash-count ht-lenguajes)
4

> ht-lenguajes
#hash(("python" multiparadigma módulos interpretado)
      ("racket" funcional módulos interpretado)
      ("c" estructurado bibliotecas compilado)
      ("java" oo paquetes compilado))
```

11.11. Void

Eventualmente, necesitamos construir funciones *que no devuelvan nada*, sino que sólo queremos que se ejecuten por sus efectos colaterales, como las funciones `display`, `printf`, y

otras. En esos casos, utilizamos el procedimiento especial `void` , que devuelve el objeto especial `#<void>`:

```
> (void)
> (begin
    "Este bloque no devuelve nada..."
    (void)
    )
> void
#<procedure:void>


> ;;; También sirve cuando sólo queremos la parte verdadera (o falsa) de un
     if: ;;;;;;;;;;;
> (define (mostrar-si-es-entero n)
    (if (integer? n)
        (printf "El parámetro es un entero\n")
        (void)
        )
    )
> (mostrar-si-es-entero 3)
El parámetro es un entero
> (mostrar-si-es-entero 5.7)
> (mostrar-si-es-entero "hola")
> (mostrar-si-es-entero -3)
El parámetro es un entero

> (void? (printf )
#t

> ;;;Compare estas dos ejecuciones: ;;;;;;;;;;;;;;;;;;;;;;
> (for-each display '(1 2 3))
123
> (map display '(1 2 3))
123(#<void> #<void> #<void>)
```

12 Expresiones y Definiciones Avanzadas

Aquí se discute sobre otras formas *avanzadas* del lenguaje Scheme para construir expresiones y para definir funciones.

12.1. La función `apply`

La sintaxis para la llamada de funciones, `(<expresión-función> <parámetro>*)`, soporta cualquier número de parámetros, pero una llamada específica siempre especifica un número fijo de parámetros reales. Como resultado, no se puede pasar directamente una lista de argumentos a una función:

```
>  (define (promedio L) ;;;;; no va a funcionar
     (/ (+ L) (length L)))

> (promedio '(1 2 3))
+: expects argument of type <number>; given (1 2 3)
```

La función `+` espera los parámetros uno por uno, no en una lista. En su lugar, utilizamos la función `apply`:

```
> (define (promedio L) ;; Esta sí va a funcionar
     (/ (apply + L) (length L)))

> (promedio '(1 2 3))
2

> (promedio '(1 2 3 4 5))
3
```

Su sintaxis es: `(apply <función> (list <parámetro-0> <parámetro-1> ...))`.

Y el resultado equivale a: `(<función> <parámetro-0> <parámetro-1> ... )`.

12.2. Bloques `lambda`

Recordemos primero que la forma básica de los bloques `lambda` es:
`( lambda ( <parámetro-formal>* ) <expresión>+ )`

Un bloque `lambda` con n parámetros formales, acepta n parámetros reales. Por ejemplo:

```
> ((lambda (x) (number->string x)) 3)
"3"

> ((lambda (x y) (+ x y)) 10 20)
30

> ((lambda (x y) (+ x y)) 10)
#<procedure>: expects 2 arguments, given 1: 10
```

12.2.1. Funciones con cualquier número de parámetros

Los bloques `lambda` también tiene la sintaxis opcional:
`( lambda <lista-de-parámetros> <expresión>+ )`
Donde `<lista-de-parámetros>` es un identificador –que no va encerrado entre paréntesis– que contendrá una lista con todos los parámetros reales pasados a la función:

```
> ((lambda x x) 1 2 3)
(1 2 3)

> ((lambda x x))
()

> (define mayor-valor-absoluto
    (lambda lista
      (apply max (map abs lista))
      )
    )

> (mayor-valor-absoluto -5 -4 -3 -2 -1 0 1 2 3)
5
```

12.2.2. Funciones con un mínimo número de parámetros

Se puede también, definir que una función tenga un mínimo número de parámetros obligatorios, pero sin máximo. La sintaxis es:

`( lambda ( <parámetro-formal>+ . <lista-de-parámetros> ) <expresión>+ )`

Todos los `<parámetro-formal>` son identificadores a los que se les asignarán –obligatoriamente– los primeros valores de los parámetros reales pasados a la función. `<lista-de-parámetros>` es un identificador que será una lista con todos los parámetros reales restantes, si los hubiera.

Ejemplo:

```
> (define mayor-valor-absoluto
    (lambda (primero . lista)
      (apply max (map abs (cons primero lista)))
```

```
      )
    )

> (mayor-valor-absoluto )
procedure mayor-valor-absoluto: expects at least 1 argument, given 0

> (mayor-valor-absoluto -5 -4 -3 -2 -1 0 1 2 3)
5
```

12.2.3. Funciones con parámetros opcionales

La sintaxis de los bloques `lambda` se puede ampliar para permitir parámetros opcionales:

```
( lambda <parámetros-formales> <expresiones-cuerpo>+ )

<parámetros-formales> ::= ( <parámetro>* ) |
                          <lista-de-parámetros> |
                          ( <parámetro>+ . <lista-de-parámetros> )
<parámetro> ::= <identificador-de-parámetro> |
                [ <identificador-de-parámetro> <valor-por-defecto> ]
```

Un parámetro de la forma `[<identificador-de-parámetro> <valor-por-defecto>]` es opcional. Cuando el argumento no es indicado en la llamada, la expresión `<valor-por-defecto>` produce un valor que se asigna como parámetro real. Esta expresión puede hacer referencia a cualquier parámetro precedente. Y todos los parámetros siguientes a uno opcional, deben ser opcionales; no se puede definir un parámetro obligatorio después de uno opcional en una misma función.

Ejemplos:

```
> (define saludar
    (lambda (nombre [apellido "Pérez"])
      (string-append "Hola, " nombre " " apellido)))

> (saludar "Pedro")
"Hola, Pedro Pérez"

> (saludar "Pedro" "Martínez")
"Hola, Pedro Martínez"


> (define saludar
    (lambda (nombre [apellido (if (equal? nombre "Juan")
                                  "Pérez"
                                  "Martínez")])
      (string-append "Hola, " nombre " " apellido)))

> (saludar "Pedro")
"Hola, Pedro Martínez"
```

```
20
21 > (saludar "Juan")
22 "Hola, Juan Pérez"
23
24 > (saludar "Eduardo" "Navas")
25 "Hola, Eduardo Navas"
```

12.2.4. Funciones con parámetros con nombre

La sintaxis de los bloques `lambda` es aún más ámplia, y puede incluír *parámetros con nombre*, o según la nomenclatura de Scheme, **parámetros de palabra clave** (véase la sección 11.7):

```
( lambda <parámetros-formales> <expresiones-cuerpo>+ )

<parámetros-formales> ::= ( <parámetro>* ) |
                          <lista-de-parámetros> |
                          ( <parámetro>+ . <lista-de-parámetros> )
<parámetro> ::= <identificador-de-parámetro> |
                [ <identificador-de-parámetro> <valor-por-defecto> ] |
                <palabra-clave> <identificador-de-parámetro> |
                <palabra-clave> [ <identificador-de-parámetro> <valor-por-
                    defecto> ]
```

Un parámetro especificado como `<palabra-clave> <identificador-de-parámetro>` es pasado a la función usando la misma `<palabra-clave>`. La posición del binomio `<palabra-clave>` e `<identificador-de-parámetro>` en la lista de parámetros reales no importa para hacer la correspondencia con los parámetros formales, ya que se asignará por correspondencia de la *palabra clave* en lugar de correspondencia de la posición.

En este momento vale la pena decir que existen cuatro tipos de parámetros:

1. *Los parámetros obligatorios por posición.* En este caso, el parámetro real y el formal se corresponden por la posición de ambos.
2. *Los parámetros opcionales por posición.* En este caso, el parámetro real, si está, se corresponde por la posición con el formal.
3. *Los parámetros obligatorios por palabra clave.* Con estos, el parámetro real, debe ir precedido por una *palabra clave* y se corresponderá con el parámetro formal que esté precedido por esa misma *palabra clave*.
4. *Los parámetros opcionales por palabra clave.* En este caso, el parámetro real, si se indica, se corresponde con el formal por la *palabra clave*.

También, hay que agregar que una vez que se define un parámetro opcional (por posición o por palabra clave), los parámetros siguientes deben ser todos opcionales.

Ejemplos:

```
> (define saludar
    (lambda (nom #:apellido ape)
      (string-append "Hola, " nom " " ape)))

> (saludar "Eduardo" #:apellido "Navas")
"Hola, Eduardo Navas"

> (saludar #:apellido "Navas" "Eduardo")
"Hola, Eduardo Navas"


> (define saludar
    (lambda (#:saludo [sal "Hola"] nom #:apellido [ape "Pérez"])
      (string-append sal ", " nom " " ape)))

> (saludar "Juan")
"Hola, Juan Pérez"

> (saludar "Karl" #:apellido "Marx")
"Hola, Karl Marx"

> (saludar "Juan" #:saludo "¿Qué ondas?")
"¿Qué ondas?, Juan Pérez"

> (saludar "Eduardo" #:apellido "Navas" #:saludo "Bonan Matenon")
"Bonan Matenon, Eduardo Navas"
```

12.2.5. Funciones con aridad múltiple

Otra forma de definir funciones con aridad variable, pero con un número finito de parámetros formales, es con el bloque `case-lambda`, que crea una función que puede tener un comportamiento completamente diferente dependiendo del número de parámetros reales que le sean pasados. Un bloque `case-lambda` tiene la sintaxis:

```
(case-lambda
  [ <parámetros-formales-case> <expresiones-cuerpo>+ ]*
  )

<parámetros-formales-case> ::= ( <identificador-parámetro>* ) |
                               <lista-de-parámetros> |
                               ( <identificador-parámetro>+ . <lista-de-
                                   parámetros> )
```

Donde cada bloque `[ <parámetros-formales-case> <expresiones-cuerpo>+ ]`, es equivalente a `(lambda <parámetros-formales-case> <expresiones-cuerpo>+)`

Al llamar una función definida por un `case-lambda` es como aplicar un `lambda` para el primer caso en que coincida el número de parámetros reales con los formales.

Cabe aclarar que el bloque `case-lambda` sólo soporta parámetros obligatorios por posición ni parámetros por palabra clave (ni obligatorios ni opcionales).

Ejemplo:

```
> (define función-de-múltiple-aridad
    (case-lambda
      [(x) "un parámetro"]
      [(x y) "dos parámetros"]
      [(x y z . w) "al menos tres parámetros"]))

> (función-de-múltiple-aridad 1)
"un parámetro"

> (función-de-múltiple-aridad 1 2)
"dos parámetros"

> (función-de-múltiple-aridad 1 2 3)
"al menos tres parámetros"

> (función-de-múltiple-aridad 1 2 3 4)
"al menos tres parámetros"
```

12.2.6. Consultando la aridad de las funciones

Cuando escribimos funciones que reciben funciones como parámetros, es necesario verificar si la aridad de las últimas es válida para el propósito de nuestra función. Para ello, Scheme provee funciones de manipulación y consulta de información de funciones. Entre ellas, podemos mencionar a `procedure-arity`, y `procedure-arity-includes?`. También podemos mencionar a la estructura `arity-at-least`.

`arity-at-least`

Esta estructura tiene la definición (véase el capítulo 13):
`(define-struct arity-at-least (value))`, donde `value` es un entero no negativo.

Una instancia `a` definida como `(define a (make-arity-at-least <num>))` indica que una función/procedimiento acepta al menos `<num>` parámetros reales.

Esto puede sonar muy extraño en este momento, pero no es tan complicado, así que mejor siga leyendo el resto de la sección.

`procedure-arity`

La función `procedure-arity` devuelve información sobre la aridad de un procedimiento. Su sintaxis es: `(procedure-arity <función>)`. Devuelve una de tres cosas:

- Un entero no negativo, lo que significa que `<función>` acepta ese número de parámetros únicamente.
- Una instancia de la estructura transparente `arity-at-least`, lo que significa que `<función>` acepta un mínimo número de parámetros, y ese mínimo es el valor (entero no negativo) del campo `value` de la estructura devuelta.
- Una lista de enteros no negativos e instancias de `arity-at-least`, lo que significa que `<función>` acepta cualquier número de parámetros que coincidan con uno de los elementos de la lista.

Ejemplos:

```
> (procedure-arity cons)
2

> (procedure-arity list)
#(struct:arity-at-least 0)

> (arity-at-least-value (procedure-arity list))
0

> (arity-at-least-value (procedure-arity (lambda (x . y) x)))
1

> (procedure-arity
   (case-lambda
     [(x) "un parámetro"]
     [(x y) "dos parámetros"]
     [(x y . z) "al menos dos parámetros"]))
(1 2 #(struct:arity-at-least 2))
```

`procedure-arity-includes?`

Tiene la sintaxis: `(procedure-arity-includes? <función> <k>)`, donde `<k>` es un entero no negativo. Esta función responde si `<función>` acepta `<k>` parámetros.

Ejemplos:

```
> (procedure-arity-includes? cons 2)
#t
> (procedure-arity-includes? cons 3)
#f
> (procedure-arity-includes?
   (case-lambda
     [(x) "un parámetro"]
     [(x y) "dos parámetros"]
     [(x y . z) "al menos dos parámetros"])
   3)
#t
```

```
> (procedure-arity-includes?
   (case-lambda
     [(x) "un parámetro"]
     [(x y) "dos parámetros"]
     [(x y . z) "al menos dos parámetros"])
   10)
#t
```

Este es un ejemplo en el que se valida la aridad de una función pasada como parámetro:

```
> (define (componer-función-unaria f x)
    (if (and (procedure? f)
             (procedure-arity-includes? f 1))
        (f (f x))
        "El primer parámetro proporcionado, no es una función unaria"))

> (componer-función-unaria sqrt 16)
2

> (componer-función-unaria cons 16)
"El primer parámetro proporcionado, no es una función unaria"

> (componer-función-unaria sqr 3)
81
```

12.3. Resultados múltiples

Una expresión normalmente produce un único resultado, pero algunas expresiones pueden producir múltiples resultados. Por ejemplo, en Scheme, las funciones `quotient` y `remainder` producen un único valor, pero la función `quotient/remainder` produce los mismos dos valores al mismo tiempo:

```
> (quotient 13 3)
4

> (remainder 13 3)
1

> (quotient/remainder 13 3)
4
1
```

Visualmente, los dos valores aparecen en líneas diferentes; algorítmicamente hablando, esto en consistente con el hecho que los algoritmos puede producir "múltiples" valores de salida, así como pueden tomar múltiples valores de entrada.

12.3.1. `values`

La función `values` acepta cualquier cantidad de parámetros y "los devuelve todos":

```
> (values)

> (values "a" 1 #\a)
"a"
1
#\a
```

Eventualmente nuestras funciones deben devolver dos o más valores simultáneamente. En esos casos, se podría optar por devolver una lista o vector con los valores correspondientes; pero usar la función `values` es más elegante y algorítmicamente más apropiado, porque con `values` se devuelven los resultados del cálculo y no una "lista" con los resultados.

A diferencia de otros lenguajes de programación, que fuerzan al programador a "devolver un sólo valor" (considere lenguajes como *C* o *Java*), el lenguaje Racket permite acercar –de nuevo– el código del programa a su representación matemática.

12.3.2. `define-values`

El bloque `define-values` asigna múltiples identificadores al mismo tiempo producidos por múltiples resultados de una única expresión:
`(define-values ( <identificador>* ) <expresión> )`

El número de resultados de `<expresión>` debe coincidir con el número de identificadores.

```
> (define-values (cociente residuo) (quotient/remainder 101 50))

> cociente
2

> residuo
1
```

12.3.3. `let-values`, y `let*-values`

De la misma manera que `define-values` asigna múltiples resultados en una definición, `let-values` y `let*-values` asignan múltiples resultados localmente:

```
> (define (muestra-cociente-y-residuo n1 n2)
    (let-values ([(cociente residuo) (quotient/remainder n1 n2)])
      (printf "El cociente es: ~a\n" cociente)
      (printf "El residuo es: ~a\n" residuo)
      )
    )
```

```
7
8  > (muestra-cociente-y-residuo 102 25)
9  El cociente es: 4
10 El residuo es: 2
```

La diferencia entre `let-values` y `let*-values` es la misma que entre `let` y `let*`: `let-values` hace asignaciones en paralelo y `let*-values` hace asignaciones secuencialmente.

12.4. Asignaciones

En ciertos casos *desesperados*, es posible considerar la asignación de nuevos valores a variables ya existentes[1]. Esto se hace con las funciones `set!` y `set!-values`.

La sintaxis es:
`(set! <identificador> <expresión>)` y
`(set!-values ( <identificador>* ) <expresión>)`

Los identificadores deben haber sido asignados previamente, por lo que no sirven para inicializar variables.

Es pertinente hacer la aclaración que el abuso de las asignaciones puede producir resultados inesperados (pero no erroneos), debido a que las asignaciones directas no son propias del paradigma funcional.

[1] Como ya habrá notado el lector, las asignaciones de variables, no suelen necesitarse en el paradigma funcional.

Ejercicios de Expresiones y Definiciones Avanzadas

1. Elaborar una función, a la cual si se le pasa de parámetro un número real N retornar su raíz cuadrada, pero si se le pasa de parámetro un número N y como segundo parámetro otro número entero positivo X indicando el grado de la raíz que se le sacará a N. Es decir: $\sqrt[X]{N}$.
2. Elaborar una función cuya restricción es que tiene que recibir parámetros indefinidos es decir el número de parámetros puede ser variable, para este ejercicio se pide que si se recibe un número tiene que retornar el número, si recibe dos números tiene que retornar el mayor de ambos, si son tres números retornar las dos raíces de la ecuación cuadrática, en la que cada número será el coeficiente literal de $Ax^2 + Bx + C = 0$.
3. Elaborar una función que reciba de parámetro una cadena y un carácter, este carácter será el que se busque en la cadena y por cada ocurrencia de este carácter en la cadena se sustituirá por un espacio en blanco. La restricción de este ejercicio es que la cadena y el carácter pueden ser pasados de parámetro en diferente orden y la función deberá funcionar correctamente.
4. Elaborar una función que reciba de parámetro una lista de símbolos que representen los atributos de un automóvil y una lista de símbolos con los valores de estos atributos. La función retornará una lista que contenga pares, cada par contendrá símbolos, indicando su atributo y su valor.

   Ejemplo: Si ingresamos lo siguiente:
   ```
   > (automovil '(Hatchback Suzuki Forza1 Rojo si Manual) '(Tipo Marca Modelo Color A/C Transmisión))
   ```

   el resultado será:
   ```
   ( (Tipo . Hatchback) (Marca . Suzuki) (Modelo . Forza1) (Color . Rojo) (A/C . si) (Transmisión . Manual))
   ```
5. Elaborar una función que reciba una lista variable de parámetros. >Si a la función no se le pasa ningún parámetro, debe retornar una lista vacía. >Si sólo se le pasa de parámetro un vector de números enteros, retornar el vector ordenado de forma ascendente. >Si el único parámetro no es vector o es vector pero no contiene números enteros, retornar #f. >Si se le pasa de primer parámetro un vector de números enteros

y de segundo parámetro un número entero, la función ingresará el número dentro del vector y deberá retornar el vector ordenado ascendentemente. >Si se le pasa como primer parámetro un vector de números enteros, y como segundo y tercer parámetro dos números enteros, la función deberá buscar en el vector el número pasado como segundo parámetro en el vector y sustituirlo por el número pasado como tercer parámetro, y deberá retornar el vector ordenado de forma ascendente. *Si hay más parámetros o si los parámetros son incorrectos entonces se mostrará un mensaje de error indicando que los parámetros son incorrectos.

6. Elaborar una función que reciba dos listas como parámetro, la primer lista que deberá recibir contendrá símbolos que correspondan a los atributos de una persona. Esto podría ser así: `'(nombre apellido edad sexo estado-civil teléfono dui nit)` (queda a libertad el número de atributos). Como segundo parámetro deberá recibir una lista que contenga los valores para cada atributo proporcionado en la primer lista. Notar que las listas deben ser del mismo tamaño y que todos los datos de la segunda lista deben ser cadenas a excepción de la edad que está en la posición tres que es un número entero y positivo. La función deberá retornar una lista de pares que contenga el atributo y su valor.
7. Elaborar una función que reciba de parámetro un número indefinido de parámetros, con la única restricción que los parámetros deberán ser sólo números enteros y positivos. Si la función no recibe parámetro alguno, entonces deberá retornar una lista vacía, si recibe un solo parámetro entonces deberá de retornar el parámetro, si recibe dos parámetros deberá retornar un par con esos dos valores, si recibe tres parámetros entonces deberá retornar el número mayor, si recibe cuatro parámetros retornar el número menor, si recibe cinco o más parámetros deberá retornar un vector de los elementos ordenados de menor a mayor.
8. Elaborar una función que reciba de parámetro a lo sumo tres parámetros, que representen un conjunto de coordenadas (x,y), estas coordenadas serán pasadas de parámetros en formato de "pares" es decir `'(x . y)` si no se recibe parámetro alguno entonces retornar una lista vacía, si hay un solo parámetro retornar el punto en el plano cartesiano en forma de "par", si recibe dos puntos retornar la distancia entre los dos puntos, si son tres puntos retornar el área del triángulo formado.
9. Elaborar una función que reciba 2 números enteros A y B que retorne una lista de pares donde la primera posición será el número y la segunda una cadena con "si" o "no" que indicará si el número es primo o no, se tomarán todos los números comprendidos en el rango $[A, B]$, la restricción de este ejercicio es que los parámetros A y B puedan ser pasados en cualquier orden.

13 Tipos de dato definidos por el programador

Aquí hablaremos sobre cómo definir variables compuestas por varios campos. Sobre Objetos y clases, léase el capítulo 18.

13.1. Estructuras simples

La sintaxis básica para declarar estructuras es:
`( define-struct <nombre-estructura> (<nombre-campo>* ) )`

Con esta definición, Scheme también crea una serie de funciones adicionales para poder manipular las estructuras de ese nuevo tipo:

- `make-<nombre-estructura>` es una función constructora que sirve para crear estructuras del nuevo tipo, y toma tantos parámetros como campos tenga el tipo.
- `<nombre-estructura>?` es una función lógica que verifica si el resultado de una expresión es del nuevo tipo.

`<nombre-estructura>-<nombre-campo>` son una serie de funciones que devuelven el valor de un campo de un elemento del nuevo tipo.

Por ejemplo:

```
1  > (define-struct punto (x y))
2
3  > (define mi-punto (make-punto 1 2))
4
5  > mi-punto
6  #<punto>
7
8  > (punto? "punto")
9  #f
10
11 > (punto? mi-punto)
12 #t
13
14 > (punto-x mi-punto)
15 1
16
```

```
> (punto-y mi-punto)
2
```

Por defecto, las estructuras creadas así son *inmutables*. Por lo que existe una función para copiar estructuras y opcionalmente actualizar algunos campos en la nueva copia. Su sintaxis es la siguiente:

```
( struct-copy <nombre-estructura> <expresión-de-estructura>
    { [ <nombre-campo> <expresión-para-un-campo> ] }*
)
```

La `<expresión-de-estructura>` debe producir una instancia del tipo `<nombre-estructura>`. El resultado de la función string-copy es una nueva instancia de `<nombre-estructura>` que es idéntica a la producida por `<expresión-de-estructura>`, excepto que sus campos indicados en los corchetes tienen el valor correspondiente al resultado de la expresión indicada.

Otra cosa importante de destacar es que no hay una verificación semántica –ni de tipo– de los valores que se asignan a los campos de las estructuras.

Ejemplo:

```
> (define p1 (make-punto 1 2))
> (define p2 (struct-copy punto p1 [x 3]))

> (punto-x p2)
3
> (punto-y p2)
2

> (define p3 (struct-copy punto (make-punto 10 20) [y 5]))

> (punto-x p3)
10
> (punto-y p3)
5
```

13.2. Estructuras derivadas

Una forma extendida de `define-struct` puede ser usada para definir un subtipo de estructura, que es un tipo de estructura que extiende a otro tipo, o que se deriva de otro.

La sintaxis es:
`(define-struct (<nombre-estructura> <estructura-madre>) ( <nombre-campo>* ) )`

La `<estructura-madre>` debe ser el nombre de la estructura a la que `<nombre-estructura>` extiende. Por ejemplo:

```
> (define-struct punto (x y))

> (define-struct (punto3d punto) (z))

> (define p (make-punto3d 10 9 8))

> p
#<punto3d>

> (punto? p)
#t

> (punto3d? p)
#t

> (punto3d-x p)
10

> (punto-x p)
10

> (punto3d-z p)
8
```

Vale la pena mencionar que cuando se trata de estructuras derivadas, hay una convención de nombres en Scheme:

> Si hay una estructura madre llamada `base`, y de esta se deriva otra llamada `derivada`, el nombre formal de esta, *debería* ser `base:derivada`. Y si de esta, a su vez, se deriva otra llamada `descendiente`, su nombre formal *debería* ser `base:derivada:descendiente`, así:

```
(define-struct base (...))
(define-struct (base:derivada base) (...))
(define-struct (base:derivada:descendiente base:derivada) (...))
```

Esta convención permite rastrear la jerarquía de estructuras, en los casos en los que amerite. Para el caso de Racket, las estructuras de casi todas las Excepciones (ver capítulo 16), se derivan de una excepción madre llamada `exn`. Así, por ejemplo, la excepción lanzada cuando se intenta dividir por cero se llama `exn:fail:contract:divide-by-zero`. Con este nombre, es posible rastrear la jerarquía de derivación de la excepción, que a su vez permite –con un poco de práctica por parte del programador– entender la clasificación de esta, sólo con su nombre.

13.3. Estructuras transparentes y opacas

Cuando una estructura se define de la forma
`( define-struct <nombre-estructura> (<campo>* ) )`
por defecto es **opaca**, es decir, que cuando se imprime, sólo se muestra el nombre del tipo de la estructura a la que corresponde. En cambio si fuera **transparente**, se imprime como un vector, mostrando el contenido de los campos.

La sintaxis para definir un tipo de estructura *transparente* es:
`( define-struct <nombre-estructura> (<nombre-campo>* ) #:transparent)`

La diferencia en la definición, es una palabra clave que se agrega después de los campos.

Ejemplo:

```
1 > (define-struct persona (nombre dirección teléfono) #:transparent)
2
3 > (define eduardo (make-persona "Eduardo NAVAS" "Antiguo Cuscatlán"
      "2210-6600, ext 1048"))
4
5 > eduardo
6 #(struct:persona "Eduardo NAVAS" "Antiguo Cuscatlán" "2210-6600, ext 1048")
```

La razón por la que las estructuras son por defecto opacas, es para proveer mayor *encapsulamiento* a las bibliotecas que se implementen con Scheme.

13.4. Estructuras mutables

Si eventualmente se requiriera de una estructura cuyos campos deban ser alterados, el tipo de estructura debe declararse como mutable, así:
`( define-struct <nombre-estructura> (<nombre-campo>* ) #:mutable)`

Al declararse así un tipo de estructura, se crean también una serie extra, de funciones con el nombre `set-<nombre-estructura>-<nombre-campo>!`; se crea una por cada campo de la estructura.

Ejemplo:

```
1  > (define-struct punto (x y) #:mutable)
2
3  > (define p (make-punto 2.5 3.6))
4
5  > p
6  #<punto>
7
8  > (set-punto-x! p 10)
9
10 > (punto-x p)
11 10
```

Cabe recalcar que un tipo de estructura puede ser declarado como *mutable* y como *transparente*:

```
> (define-struct persona (nombre dirección teléfono) #:mutable #:
    transparent)

> (define eduardo (make-persona "Eduardo" "El Salvador" "2210-6600"))

> eduardo
#(struct:persona "Eduardo" "El Salvador" "2210-6600")

> (set-persona-nombre! eduardo "Edgardo")

> eduardo
#(struct:persona "Edgardo" "El Salvador" "2210-6600")
```

Ejercicios de Tipos definidos por el programador

1. Crear una función que pida en tiempo de ejecución un número entero positivo que indique el día, un número entero positivo que represente un mes, y un número entero positivo que represente un año, y retornar una estructura de tipo fecha, la definición de la estructura es la que se muestra a continuación:
   `(define-struct fecha (día mes año) #:transparent)`
2. Elaborar una función que pida en ejecución tres puntos del plano cartesiano, y cada punto será una estructura de tipo "punto" con campos X y Y respectivamente , luego retornará una lista con los 3 puntos ordenados de menor a mayor bajo el criterio de la distancia al origen.
3. Elaborar una función que reciba como parámetro una lista de estructuras de tipo fecha, y que retorne la lista ordenada de fecha anterior a posterior. Si algún elemento que esté en la lista no es de tipo fecha, retornar una lista nula.
4. Elaborar una función que reciba una cadena que corresponda a un nombre de persona, y un número indicando la edad de la persona, y como último parámetro el número de DUI(cadena) de la persona, la función deberá de retornar una estructura transparente de tipo persona, pero con la condición que si la persona es menor de edad el campo de DUI estará vacío pudiéndose modificar en el futuro.
5. Elabore una función que reciba como parámetro una estructura de tipo persona, la función deberá de retornar una estructura de tipo empleado (derivada de persona) y para ello se deberá de capturar en tiempo de ejecución el NIT(cadena) y el número de teléfono (cadena).
6. Elaborar un función que reciba como parámetro una lista de estructuras de tipo persona, dicha función deberá retornar la lista de estructuras ordenada alfabéticamente por nombre de persona. Note que nombres como "Óscar", van junto con los nombres con letra inicial "o" y no antes de los nombres con inicial "a", ni después de los nombres con inicial "z".
7. Elaborar una función que reciba de parámetro un conjunto de puntos en forma de pares, la función deberá de retornar un vector ordenado de mayor a menor distancia al origen, que contenga los puntos pero con una estructura `puntoD`, la cual tiene en un campo el par `(x . y)` y en el otro campo la distancia d de ese punto al origen.

14 Módulos Funcionales

Así como en otros lenguajes de programación, en Scheme se pueden definir **Módulos Funcionales**. Estos *módulos* equivalen a las *bibliotecas de funciones* o *bibliotecas de objetos/-clases* de otros lenguajes.

En general un *módulo* sirve para encapsular cierto código, sin que el usuario del mismo tenga que conocer los detalles de implementación. En el caso de Racket, se trata de encapsular definiciones de funciones, de constantes, de estructuras, de clases, etc.

Por defecto, un archivo de código Scheme, que comience con la línea: `#lang racket` es un módulo funcional, cuyo nombre es el nombre del archivo, sin la extensión `.rkt` (`.ss` o `.scm`).

Por defecto, todas las definiciones de un módulo, son *privadas*. Y para que sean útiles para otros usuarios/programadores, hay que hacerlas *públicas*.

Por ejemplo:

```
1  #lang racket
2  ;pastel.rkt
3
4  ;; Vuelve 'pública' la definición de 'imprimir-pastel' y 'número-por-
       defecto':
5  (provide imprimir-pastel número-por-defecto)
6
7  ;;Estas definiciones son invisibles fuera de este módulo
8  (define flama #\.)
9  (define candela #\|)
10 (define pan #\x)
11 (define base #\-)
12
13 (define número-por-defecto 3)
14
15 ; Dibuja un pastel con n candelas
16 (define imprimir-pastel
17   (lambda ([n número-por-defecto])
18     (if (and (integer? n) (exact-nonnegative-integer? n))
19         (begin
20           (printf "   ~a  \n"  (make-string n flama))
21           (printf " .-~a-.\n"  (make-string n candela))
22           (printf "_|x~ax|_\n" (make-string n pan))
23           (printf "---~a---\n" (make-string n base))
24           )
25         (error "Se espera un número entero no negativo")
26         )))
```

Entonces, desde otro archivo en el mismo directorio, se puede invocar la función `imprimir-pastel`, por ejemplo desde el siguiente archivo:

```
#lang racket
;mostrador-de-pasteles.rkt

(define secuencia-de-enteros
  (lambda (num-elementos [inicio 0] [paso 1])
    (define (aux i contador lista)
      (if (>= contador num-elementos)
          (reverse lista)
          (aux (+ i paso) (add1 contador) (cons i lista))
      ))
    (if (and (exact-nonnegative-integer? num-elementos)
             (integer? inicio)
             (integer? paso))
        (aux inicio 0 empty)
        (error "Error en los parámetros")
        )
    ))

(require "pastel.rkt")

;Imprime un pastel de 'número-por-defecto' candelas:
(imprimir-pastel)
(printf "El número por defecto de candelas es: ~a\n" número-por-defecto)

;Imprime 8 pasteles desde 0 hasta 7 candelas:
(for-each imprimir-pastel (secuencia-de-enteros 8))
```

Para ejecutarlo, evaluamos el comando:
`$ racket mostrador-de-pasteles.rkt`

Otro detalle importante, es que cuando se invoca (con `require`) a un módulo funcional, este es ejecutado, de tal manera que se realizan todas las definiciones en él, y se ejecutan todas las expresiones que contenga.

14.1. Visibilizando definiciones de estructuras

Si en el archivo `biblioteca.rkt` tuvieramos la definición
`(define-struct estructura (campo1 campo2)`, y quisieramos hacer visible la definición de la estructura, debemos agregar un bloque especial en la forma `provide`:
`(provide ... (struct-out estructura) ... )`

Y con ello, disponemos de las funciones de manipulación para la estructura (`make-estructura`, `estructura?`, `estructura-campo1`, etc.) en otro módulo que importe a `biblioteca.rkt`.

15 Entrada y Salida

Aquí se describe lo básico para comprender las posibilidades de **Entrada y Salida** en Scheme.

En Scheme, un **puerto** representa un flujo de entrada o de salida, como un archivo, una terminal, una conexión TCP o una cadena en memoria. Más específicamente un **puerto de entrada** representa un flujo desde el cual un programa puede leer datos y un **puerto de salida** representa un flujo que un programa puede usar para escribir datos.

15.1. Imprimir datos

En Scheme hay dos formas de imprimir valores de tipos primitivos:

- `write`. Esta función imprime un valor en la misma manera en que este se representa en el lenguaje, que es la misma forma en que se muestran los valores en el entorno interactivo.

- `display`. Esta función tiende a reducir un valor a su representación de bytes o de caracter. Es una forma muy legible de mostrar los datos, pero es menos precisa sobre el tipo de dato que se muestra.

He aquí algunas comparaciones sobre el comportamiento de ambas:

write	display
> (write 1/2) 1/2 > (write #\x) #\x > ;Note las comillas en la salida > (write “hola”) "hola" > (write #"nos vemos") #"nos vemos" > (write '|símbolo partido|) |símbolo partido| > (write '("cadena" símbolo)) ("cadena" símbolo) > (write write) #<procedure:write>	> (display 1/2) 1/2 > (display #\x) x > ;No hay comillas en la salida > (display "hola") hola > (display #"nos vemos") nos vemos > (display '|símbolo partido|) símbolo partido > (display '("cadena" símbolo)) (cadena símbolo) > (display write) #<procedure:write>

Finalmente, la función `printf` formatea una cadena de texto con el contenido de otros valores. Estos otros valores, se ingresan como parámetros extra en la función, y en la cadena de formateo, como `~a` o `~s`. Poner `~a` provoca que el parámetro correspondiente sea agregado con display, y poner `~s` provoca que sea agregado con write:

```
> (define (prueba-de-printf valor)
    (printf "Con display: ~a\nCon write: ~s\n" valor valor))

> (prueba-de-printf "hola")
Con display: hola
Con write: "hola"

> (prueba-de-printf #\r)
Con display: r
Con write: #\r

> (prueba-de-printf #"cadena ascii")
Con display: cadena ascii
Con write: #"cadena ascii"

> (prueba-de-printf #("vector" "con números" 3))
Con display: #(vector con números 3)
Con write: #("vector" "con números" 3)
```

Resumiendo:
`~a -> display`
`~s -> write`

15.2. Leer datos

Existen muchos mecanismos para leer datos en Scheme. De hecho, "el lector" de Scheme (*The Reader*), es un *analizador léxico y sintáctico desdendente recursivo*, es decir, un programa procesador bastante avanzado.

15.2.1. Lectura "básica"

Existen siertas funciones básicas, muy típicas de un lenguaje de alto nivel como Scheme. En todos los siguientes casos, el puerto de lectura/entrada es opcional, y su valor por defecto es la entrada estándar (o, lo que es lo mismo, lo que devuelve la función `(current-input-port)` que se explica más adelante, en la sección 15.4):

- `(read-char {<entrada>}? )`: Lee un caracter en la codificación de "el lector", que por defecto es *UTF-8*.
- `(read-byte {<entrada>}? )`: Lee un byte, es decir, un código *ascii*.
- `(read-line {<entrada> {<modo>}? }? )`: Lee una cadena que va desde el punto actual del cursor hasta el próximo fin de línea. Lo que se considera como fin de línea depende del segundo parámetro opcional:
  `<modo> ::= { 'linefeed | 'return | 'return-linefeed | 'any | 'any-one }` y el valor por defecto es `'linefeed`.
  - `'linefeed` interpreta el fin de línea cuando encuentra el caracter `\n`. Este es el comportamiento correcto en sistemas Unix.
  - `'return` interpreta el fin de línea cuando encuentra el caracter `\r`. Este es el comportamiento correcto en sistemas Macintosh.
  - `'return-linefeed` interpreta el fin de línea cuando encuentra la secuencia `\r\n`. Este es el comportamiento correcto en sistemas Windows.
  - `'any` interpreta el fin de línea cuando encuentra un `\n`, `\r` o `\r\n`.
  - 'any-one interpreta el fin de línea cuando encuentra un \n o un \r.
- `(read-string <cuenta> {<entrada>}? )`: Lee una cadena de a lo sumo `<cuenta>` caracteres.
- `(read-bytes <cuenta> {<entrada>}? )`: Lee una cadena de bytes de a lo sumo `<cuenta>` bytes.

En todos los casos, pueden devolver el objeto especial `eof` que se verifica con la función `eof-object?`. La devolución de este valor, indica que se alcanzó el fin del flujo.

Los ejemplos se presentan en la página 117.

15.2.2. Lectura avanzada

Se dispone de la función `read` que tiene la sintaxis:
`(read {<entrada>}? )`

La función `read` por defecto lee *un* dato de los tipos nativos de Scheme en un sólo paso. El flujo del cual lee la función, debe seguir cierta sintaxis[1]:

- `(`, `[` o `{`, indica el inicio de un par o una lista.
- `)`, `]` o `}`, indica el cierre de una estructura previamente abierta (no necesariamente una lista, como se explica más abajo).
- `"` indica el inicio de una cadena, que se cierra en la siguiente `"`.
- `;` indica que toda esa línea es un comentario y será ignorada.
- `#t`, `#T`, `#f` o `#F`, se convierten en los respectivos valores booleanos.
- `#(`, `#[` o `#{`, indican el inicio de vectores.
- `#\` inicia un caracter, tal y como se escriben en el lenguaje.
- `#"` inicia una cadena de bytes.
- `#%` inicia un símbolo (lo mismo que si no coincide con ningún otro de estos patrones).
- `#:` inicia una palabra clave.
- `#|` inicia un bloque de comentario que será ignorado (hasta que se encuentre un `|#`).
- `#i`, `#e`, `#x`, `#o`, `#d`, `#b`, `#I`, `#E`, `#X`, `#O`, `#D`, `#B`, inicia un número inexacto, exacto, hexadecimal, octal, decimal o binario, respectivamente.
- `#hash` inicia una tabla hash.
- `,` y `` ` `` y otros símbolos y combinaciones tienen otros significados más allá del objetivo de este libro.
- `#sx` inicia una expresión de racket (*Scheme eXpression*). Esta opción es muy útil para convertir texto en código fuente. Esto se aplica en el cap Evaluación Dinámica.

Ejemplos (en todos ellos, tan sólo la última línea es la respuesta de la expresión, y todas las líneas entre la expresión y la respuesta fue introducida por el teclado):

[1] Resumida aquí

```
> (read)
3
3

> (+ 2 (read))
4
6

> (+ 1 (read))
#xf
16

> (string-append "Hola, " (read))
"Eduardo"
"Hola, Eduardo"

> (string-append "Hola, " (read))
#|este es un comentario muy largo
que tiene varias líneas y que serán ignoradas por el lector|#
"Eduardo"
"Hola, Eduardo"

> (rest (read))
(1 "cadena" ;Este es un comentario de una línea
3.45 #(1 2 3)
;otro comentario
)
("cadena" 3.45 #(1 2 3))
```

Consulte la sección The Reader (http://docs.racket-lang.org/reference/reader.html) de la documentación oficial de Racket para más detalles.

15.3. Tipos de Puerto

En cada caso, es necesario *abrir* los puertos para poder transmitir (leer o escribir) datos a través de él, y cuando ya no se necesiten, estos se deber *cerrar*. La función de apertura, depende del tipo particular de puerto que se necesite abrir, pero la función de cierre sólo depende de la dirección del flujo; en todos los casos son `close-input-port` y `close-output-port`.

También hay otras funciones que operan sobre puertos:

- `input-port?` que verifica si el parámetro indicado es un puerto de entrada/lectura.
- `output-port?` que verifica si el parámetro indicado es un puerto de salida/escritura.
- `port?` que verifica si el parámetro indicado es un puerto de entrada/lectura o salida/escritura.
- `port-closed?` que verifica si un puerto está cerrado.

- `eof-object?` verifica si el parámetro proporcionado (que debería provenir de la lectura de un flujo de entrada/lectura) indica que ya se ha acabado el flujo (puede implicar una conexión terminada o el fin de un archivo).
- `flush-output` fuerza el vaciado del buffer hacia el dispositivo de escritura.

15.3.1. Archivos

Para abrir un archivo para lectura, se utiliza la función `open-input-file`. Para abrir un archivo para escritura, se usa la función `open-output-file`. Para abrir un archivo para lectura y escritura se usa la función `open-input-output-file`.

Ejemplo:

```
1 > (define salida (open-output-file "archivo.txt"))
2 > (display "Hola" salida)
3 > (close-output-port salida)
4 > (define entrada (open-input-file "archivo.txt"))
5 > (read-line entrada)
6 "Hola"
7 > (close-input-port entrada)
```

A continuación se describe la sintaxis de las funciones de apertura de puerto de archivo:

`open-input-file`

La sintaxis completa de la función `open-input-file` es:
`(open-input-file <ruta> {#:mode { 'binary | 'text } }? )`, donde la opción por defecto para el parámetro `#:mode` es `'binary`.

`open-output-file`

La sintaxis completa de la función `open-output-file` es:
`(open-output-file <ruta>`
`{#:mode { 'binary | 'text } }?`
`{#:exists { 'error | 'append | 'update | 'can-update | 'replace | 'truncate | 'must-truncate } }? )`, donde el parámetro `#:exits` indica el comportamiento a seguir cuando el archivo indicado en `<ruta>` ya existe. Por defecto se toma `'error`. Su significado es:

- `'error` lanza una excepción cuando el archivo ya existe.
- `'append` si el archivo ya existe, el cursor se coloca al final del archivo.
- `'update` se coloca al final del archivo y genera una excepción si no existe.

- `'can-update` abre el archivo sin truncarlo (es decir, sin borrarlo), o lo crea si no existe.
- `'replace` borra el archivo, si existe, y crea uno nuevo.
- `'truncate` borra el contenido del archivo, si existe.
- `'must-truncate` borra el contenido de un archivo existente, y lanza una excepción si no existe.

`open-input-output-file`

La sintaxis completa de la función `open-input-output-file` es:
`(open-input-output-file <ruta>`
`{#:mode { 'binary | 'text } }?`
`{#:exists { 'error | 'append | 'update | 'replace | 'truncate } }? )`, donde el parámetro `#:exits` indica el comportamiento a seguir cuando el archivo indicado en `<ruta>` ya existe. Por defecto se toma `'error`. Su significado es:

- `'error` lanza una excepción cuando el archivo ya existe.
- `'append` si el archivo ya existe, el cursor se coloca al final del archivo.
- `'update` se coloca al final del archivo y genera una excepción si no existe.
- `'replace` borra el archivo, si existe, y crea uno nuevo.
- `'truncate` borra el contenido del archivo, si existe.

Ejemplo

Este es un programa que lee e imprime en pantalla todas las líneas de un archivo proporcionado como parámetro, anteponiéndoles el número de línea correspondiente:

```
1  #lang racket
2  ;lee-líneas.rkt
3
4  (define (mostrar-líneas nombre-archivo)
5    (define (aux flujo número-de-línea)
6      (let ([línea (read-line flujo)])
7        (if (eof-object? línea)
8            (close-input-port flujo)
9            (begin
10             (printf "~a: ~s\n" número-de-línea línea)
11             (aux flujo (add1 número-de-línea))
12             )
13           )
14       )
15     )
16   (if (file-exists? nombre-archivo)
```

```
17        (aux (open-input-file nombre-archivo) 1)
18        (error (string-append "No existe el archivo " nombre-archivo))
19        )
20    )
21
22  (let ([parámetros (current-command-line-arguments)])
23    (if (not (= 1 (vector-length parámetros)))
24        (error "Se espera el nombre de un archivo como parámetro")
25        (mostrar-líneas (vector-ref parámetros 0))
26        )
27    )
```

Este se ejecuta:
`$ racket lee-líneas.rkt <archivo>`

Procesamiento automatizado

Existen accesoriamente, dos funciones para realizar de manera automatizada la apertura de un archivo y pasarle el puerto abierto resultante a una función. Estas funciones son:
`(call-with-input-file <ruta> <procedimiento> {#:mode { 'binary | 'text } }? )`
y
`(call-with-output-file <ruta> <procedimiento> {#:mode { 'binary | 'text } }? {#:exists { 'error | 'append | 'update | 'replace | 'truncate } }? )`

En ambos casos, `<ruta>` es la ruta y nombre de un archivo, y `<procedimiento>` debe ser una función que reciba como único parámetro obligatorio un puerto de entrada o de salida, respectivamente.

El comportamiento es el siguiente:

1. Se intenta abrir el archivo indicado en `<ruta>` utilizando el modo y comportamiento indicado en los parámetros opcionales.
2. Una vez abierto el archivo, se ejecuta `<procedimiento>` pasándole como parámetro al puerto recién abierto.
3. Cuando `<procedimiento>` finaliza, se cierra el puerto correspondiente.
4. Finalmente, el resultado de `<procedimiento>` es el resultado de `call-with-input-file` o `call-with-output-file`.

La utilidad de estas funciones es que las funciones que contienen la lógica del procesamiento de los flujos (de entrada o de salida) no se mezclen con *la logística de abrir el flujo*, lo cual es un procedimiento que depende del tipo de puerto.

Considere la siguiente variante del código anterior:

```
1  #lang racket
2  ;lee-líneas2.rkt
```

```
(define (mostrar-líneas2 flujo-de-datos)
  (define (aux flujo número-de-línea)
    (let ([línea (read-line flujo)])
      (if (eof-object? línea)
          (void)
          (begin
            (printf "~a: ~s\n" número-de-línea línea)
            (aux flujo (add1 número-de-línea))
            )
          )
      )
    )
  (aux flujo-de-datos 1)
  )

(let ([parámetros (current-command-line-arguments)])
  (if (not (= 1 (vector-length parámetros)))
      (error "Se espera el nombre de un archivo como parámetro")
      (let ([nombre-archivo (vector-ref parámetros 0)])
        (if (file-exists? nombre-archivo)
            (call-with-input-file nombre-archivo mostrar-líneas2)
            (error (string-append "No existe el archivo " nombre-archivo))
            )
        )
      )
  )
```

15.3.2. Cadenas

Así como un archivo puede *abrirse* para leer de él o para escribir en él, también se puede hacer lo mismo con las cadenas de texto. Las funciones para abrir un puerto de cadena son: `open-input-string` y `open-output-string`. Por lo demás, funcionan igual que cualquier puerto:

```
> (define cadena-fuente "Esta es una línea de una cadena\nEsta es la
    segunda\ny la tercera.")
> (define p-lectura (open-input-string cadena-fuente))
> (read-line p-lectura)
"Esta es una línea de una cadena"
> (read-line p-lectura)
"Esta es la segunda"
> (read-line p-lectura)
"y la tercera."
> (read-line p-lectura)
#<eof>
> (close-input-port p-lectura)
>
>
> (define p-escritura (open-output-string))
```

```
> (get-output-string p-escritura)
""
> (for-each (lambda (x)
              (display x p-escritura)
              (newline p-escritura)
              )
            '(0 "1" "segundo" #"cuarto"))
> (get-output-string p-escritura)
"0\n1\nsegundo\ncuarto\n"
> (close-output-port p-escritura)
```

La función `get-output-string` toma un puerto de salida de cadena y devuelve el contenido en forma de cadena.

15.3.3. Conexiones TCP

Con Racket se pueden realizar conexiones TCP con gran facilidad. Básicamente se utilizan tres funciones:

- `tcp-listen` que abre un puerto de la computadora servidor para *escuchar* las conexiones que lleguen por ahí.
- `tcp-accept` para que un servidor *acepte* conexiones provenientes de un cliente.
- `tcp-connect` para que un cliente solicite una conexión con un servidor.
- `tcp-close` para terminar una escucha de puerto, es la contraparte de `tcp-listen`.

Ejemplo:

```
#lang racket
;comunicacion.rkt

(provide hablar-inmediato)

(define hablar-inmediato
  (lambda (puerto-salida cadena . parámetros)
    (apply fprintf (cons puerto-salida (cons cadena parámetros)))
    (flush-output puerto-salida)
    ))
```

```
#lang racket
;servidor.rkt

(require "comunicacion.rkt")

(define servidor (tcp-listen 65432)) ;; Aquí va el puerto de escucha. Falla
    si está ocupado.

(define (escuchar-aux entrada salida)
  (let ([línea (read-line entrada)])
```

```
10      (if (eof-object? línea)
11          (void)
12          (begin
13            (cond [(equal? "" línea) (void)]
14                  [(and (char=? #\¿ (string-ref línea 0))
15                        (char=? #\? (string-ref línea (sub1 (string-length
                              línea)))))
16                   ;(display "pregunta\n")
17                   (hablar-inmediato salida "Usted ha hecho una pregunta...\n
                         ")
18                   (sleep 2)
19                   (hablar-inmediato salida "Buscando la respuesta\n")
20                   (sleep 2)
21                   (hablar-inmediato salida "Lo siento, no sé la respuesta\n
                         ")
22                   ]
23                  [(equal? "ya no escuche más" línea)
24                   (display "mensaje para salir, recibido\n")
25                   (tcp-close servidor)
26                   (exit)
27                   ]
28                  [else
29                   ;(display "mensaje incomprensible\n")
30                   (hablar-inmediato salida "No entiendo lo que me dice\n")
31                   ]
32                  )
33            (escuchar-aux entrada salida)
34            )
35          )
36      )
37    )
38
39  (define (escuchar)
40    (define-values (entrada salida) (tcp-accept servidor))
41    (printf "Este es el servidor:Conexión aceptada\n")
42    (hablar-inmediato salida "Hola, este es su amigo el Servidor, ¿cómo está
          ?\n")
43    (escuchar-aux entrada salida)
44    (printf "Este es el servidor:Conexión terminada\n")
45    (escuchar)
46    )
47
48  (escuchar)
```

```
1  #lang racket
2  ;cliente.rkt
3
4  (require "comunicacion.rkt")
5
6  ;;La IP del servidor (o su nombre de dominio) y el puerto
7  (define-values (lectura escritura) (tcp-connect "localhost" 65432))
```

```
8
9  (define (comunicarse)
10   (let ([comando (read-line)])
11     (if (equal? "salir" comando)
12         (begin
13           (close-output-port escritura)
14           (exit)
15           )
16         (if (port-closed? escritura)
17          (begin
18            (printf "El puerto ya se cerró\nSaliendo...\n")
19            (exit)
20            )
21          (hablar-inmediato escritura "~a\n" comando)
22          )
23         )
24     )
25   (comunicarse)
26   )
27
28 (define (escuchar)
29   (let ([mensaje (read-line lectura)])
30     (if (eof-object? mensaje)
31         (void)
32         (begin
33           (printf "~a\n>> " mensaje)
34           (escuchar)
35           )
36         )
37     )
38   )
39
40 (void (thread escuchar))
41 (comunicarse)
```

15.4. Puertos de Entrada/Salida por defecto

Hasta ahora, se han utilizado las funciones `display`, `printf`, `write`, `read-char`, `read-line` y `read` para imprimir información en patalla. Estas funciones, utilizan por defecto tres flujos predeterminados:

- `(current-input-port)` devuelve el puerto de entrada estándar.
- `(current-output-port)` devuelve el puerto de salida estándar.
- `(current-error-port)` devuelve el puerto de salida estándar de error.

Para cambiar los puertos por defecto, se pueden usar estas mismas funciones con un parámetro puerto:

- `(current-input-port <p>)` cambia el puerto de entrada estándar a `<p>`.
- `(current-output-port <p>)` cambia el puerto de salida estándar a `<p>`.
- `(current-error-port <p>)` cambia el puerto de salida estándar de error a `<p>`.

Ejercicios de Entrada y Salida

1. Dado como parámetro el nombre de un archivo, elaborar un programa que elimine del archivo todas las letras "a".
2. Haga una variante del ejercicio anterior en el que esta vez dado de parámetro el nombre del archivo y dos caracteres, el primer carácter se buscará para ser sustituido por el segundo carácter en todo el archivo.
3. Elaborar una función que reciba de parámetro el nombre de un archivo y que retorne el número de líneas que este archivo tiene. Si el archivo no existe retornar -1.
4. Hacer un programa que dado como parámetro el nombre de un archivo de texto, aplicarle una suerte de encriptamiento, en el que cada carácter en posición par, se almacene en un archivo llamado "par.txt" y cada carácter en posición impar se almacenara en un archivo llamado "impar.txt", el archivo original será eliminado quedando sólo los dos archivos resultantes.
5. Haga un programa que desencripte un archivo encriptado con el algoritmo del ejercicio anterior, dados los archivos "par.txt" e "impar.txt".
6. Elaborar una función de encriptamiento en el que dado de parámetro el nombre de un archivo de texto, sumarle a cada carácter en posición par dos caracteres, y a cada carácter en posición impar sumarle tres posiciones. Por ejemplo, si el carácter `#\u0040` está en posición par, se deberá sustituir por el carácter `#\u0042`.
7. Construya la función de desencriptamiento correspondiente al ejercicio anterior.
8. "Se ha dejado una tarea en la que se tiene que entregar un informe de 500 palabras, el profesor tiene que calificar de forma rápida y con exactitud lo que se pidió, por eso necesita, entre otras cosas, un programa que cuente las palabras, y le pidió la ayuda a usted". Así que tiene que elaborar un programa que reciba de parámetro el nombre del archivo a revisar y tiene que mostrar el número de palabras que tiene el archivo.
9. Usando la estructura `persona` construya una función que ingrese 5 instancias de `persona` en un archivo llamado "registro.txt", una estructura por línea. Si el archivo no existe retornar un mensaje de error.
10. Construya una función que retorne `#t` si un archivo llamado "información.txt" existe y tiene contenido, es decir que no esté vacío, y `#f` en caso de que no exista o no tenga información alguna.

11. Elaborar un programa que reciba de parámetro el nombre de un archivo de texto, dicho archivo debe contener en cada línea una serie de palabras. La primera representa el nombre de un continente y las demás representan nombres de los países de ese continente, se pide leerlos del archivo y mostrarlos en la salida estándar en forma de pares que contengan una cadena con el nombre del continente y un vector de cadenas con los nombre de los países en el archivo. Los vectores deberán estar ordenados alfabéticamente y los pares deberán mostrarse también ordenados por el nombre del continente. Si el archivo no existe retornar un mensaje de error.
12. Elabore un programa que realice la función de buscar una palabra en un archivo de texto. El programa recibirá de parámetro el nombre del archivo, pero pedirá en tiempo de ejecución una palabra a buscar y si la encuentra, mostrará un mensaje indicándolo junto con la fila y la columna en la que la encontró. Si no la encontró, mostrará el mensaje correspondiente. Si el archivo no existe mostrará un mensaje de error.
13. Elaborar una función que reciba como primer parámetro el nombre de un archivo "origen.txt" y como segundo parámetro el nombre de un archivo "destino.txt". La función deberá copiar la última letra del archivo "origen.txt" en la primer posición del archivo "destino.txt", la penúltima letra del archivo "origen.txt" en la segunda posición del archivo "destino.txt" y así sucesivamente.

16 Excepciones

En Scheme, cuando sucede un error en tiempo de ejecución, se ***lanza*** una excepción. Y a menos que la excepción sea ***atrapada***, se gestionará imprimiendo en la salida estándar de error un mensaje asociado con la excepción y terminando los cálculos.

Por ejemplo:

```
> (/ 100 0)
. . /: division by zero

> (car 35)
. . car: expects argument of type <pair>; given 35

> (define p (open-input-string "Esta es la cadena fuente"))
> (read-string 5 p)
"Esta "
> (close-input-port p)
> (read-string 5 p)
. . read-string: input port is closed
```

16.1. Atrapar Excepciones

Para *atrapar* una excepción, se usa el bloque `with-handlers` que tiene la sintaxis:
`(with-handlers ( { [<f-evaluadora> <f-manejadora>] }* ) <exp-cuerpo>+ )`

Y funciona de la siguiente manera: Cuando aparece una forma como esta en el curso actual de ejecución, se comienzan a evaluar las expresiones del cuerpo, las `<exp-cuerpo>`. Si este código lanza alguna excepción, se llama la primera función `<f-evaluadora>`. Si esta se evalúa a verdadero, se ejecuta la correspondiente `<f-manejadora>` y su resultado será el resultado del bloque with-handlers. Si `<f-evaluadora>` se evalúa a falso, se probará con la siguiente `<f-evaluadora>` si la hay, y así sucesivamente. Si ninguna `<f-evaluadora>` resulta en verdadero, la excepción será relanzada para que otro bloque `with-handlers` de nivel superior la atrape (talvez).

Todas las funciones `<f-evaluadora>` y las `<f-manejadora>` deben recibir un único parámetro obligatorio que será el valor que represente a la excepción lanzada en el cuerpo. Típicamente será una instancia de alguna estructura derivada de `exn:fail` (recuérdese la sección 13.2).

Ejemplo:

```
> (with-handlers ([exn:fail:contract:divide-by-zero?
                   (lambda (e) +inf.0)])
    (/ 100 0))
+inf.0


> (with-handlers ([exn:fail:contract:divide-by-zero?
                   (lambda (e) +inf.0)])
    (car 35))
. . car: expects argument of type <pair>; given 35


> (with-handlers ([exn:fail:contract:divide-by-zero?
                   (lambda (e) +inf.0)]
                  [exn:fail? (lambda (e) (exn-message e))])
    (define p (open-input-string "Esta es la cadena fuente"))
    (display (read-string 5 p))
    (close-input-port p)
    (read-string 5 p)
    )

Esta "read-string: input port is closed"
```

16.2. Las funciones error y raise

La función `error` es una manera de crear su propia excepción, ya que toma una cadena de texto como parámetro, y la encapsula en una estructura de tipo `exn:fail`:

```
> (error "¡Error fatal!! ¡¡Todos vamos a morir!!")
. . ¡Error fatal!! ¡¡Todos vamos a morir!!

> (with-handlers ([exn:fail? (lambda (e) "¡Que no cunda el pánico!")])
    (error "¡Error fatal!! ¡¡Todos vamos a morir!!"))
"¡Que no cunda el pánico!"
```

Lo usual es que las excepciones sean instancias de la estructura `exn:fail` o de alguna de sus derivadas (lo que incluye el resultado de la función `error`), pero en Scheme, podemos lanzar nuestras propias *excepciones* que no se apeguen a esta *costumbre*.

La función `raise` nos permite lanzar cualquier objeto o valor como una excepción:

```
> (raise 2)
uncaught exception: 2
> (with-handlers ([(lambda (v) (equal? v 2)) (lambda (v) 'dos)])
    (raise 2))
dos
> (with-handlers ([(lambda (v) (equal? v 2)) (lambda (v) 'dos)]
                  [string? (lambda (e) (printf "La excepción es una cadena\
                     n"))])
```

```
8       (raise "otro error"))
9   La excepción es una cadena
```

17 Evaluación Dinámica de Código

Scheme es un lenguaje de programación **dinámico**, ya que ofrece muchas facilidades para cargar, compilar y hasta construír nuevo código en tiempo de ejecución.

17.1. La función `eval`

La función `eval` toma una expresión "*apostrofada*" (es decir, una expresión precedida por un caracter apóstrofo «`'`») y la evalúa. Otra forma de describir el parámetro es como una lista de identificadores y otros valores primitivos.

Por ejemplo:

```
> (eval '(+ 1 2))
3

> (eval (read))
(* 4 (+ 2 3))
20

> (define expresión '(+ x (* x 5)))
> (define x 2)
> (eval expresión)
12

> (cons sqrt '(4))
(#<procedure:sqrt> 4)
> (eval (cons sqrt '(4)))
2

> (define (f x) (expt 2 x))
> ((eval 'f) 4)
16
> (define símbolo 'f)
> ((eval símbolo) 5)
32
> símbolo
f
```

17.2. Creación y ejecución dinámica de código fuente

Puede utilizarse la función `eval` y `read` para convertir dinámicamente una cadena con una expresión de racket válida, en una función que evalúa esa expresión. A continuación se presentan tres ejemplos:

```
> (define expresión (read))
"(+ x (* x 5))"
> (define x (read))
2
> (define p-cad (open-input-string expresión))
> (define resultado (eval (read p-cad)))
> resultado
12
```

```
> (define cadena "(+ x 1)")
> (define función-cadena (string-append "#sx(lambda (x) " cadena ")"))
> función-cadena
"#sx(lambda (x) (+ x 1))"
> (define p (open-input-string función-cadena))
> (close-input-port p)
> (define exp (read p))
> ((eval exp) 3)
4
> ((eval exp) 1.22)
2.2199999999999998
> ((eval exp) 3.1416)
4.1416
```

```
> (define cadena "#sx(lambda (x y) (+ (sqrt (sqr x) (sqr y))))")
> (define entrada (open-input-string cadena))
> (define hipotenusa (eval (read entrada)))
> (read entrada)
#<eof>
> (close-input-port entrada)
> hipotenusa
#<procedure>
> (procedure-arity hipotenusa)
2
> (hipotenusa 3 4)
5
```

18 Programación Orientada a Objetos

En este libro no se abordarán los conceptos relacionados con el paradigma de *Programación Orientada a Objetos*, sino que se aborda cómo implementar tal programación en Scheme.

18.1. Definición de Clases

De la misma forma que un bloque `lambda` es un procedimiento sin nombre, existe el bloque `class` que es una clase sin nombre. Su sintaxis es:
`( class <superclase> <declaración-o-expresión>* )`

Y al igual que podemos utilizar `define` con `lambda` para definir funciones/procedimientos con nombre, podemos utilizar `define` con `class` para definir clases con nombre:
`(define <nombre-clase> (class <superclase> <declaración-o-expresión>* ))`

Por convención, los nombres de las clases en Scheme, terminan con el caracter `%`. La clase raíz integrada del lenguaje se llama `object%`.

18.2. Definición de Interfaces

En Scheme, las interfaces se definen así:
`( interface (<superinterface>*) <identificador>* )`

Donde `<identificador>` es un identificador que deberá ser provisto como público por la clase que implemente la interface. De no ser así, se lanzará un error al momento de evaluar la definición de la clase en cuestión.

Para definir una clase que implemente interfaces, se usa el bloque `class*`:
`( class* <superclase> (<interface>*) <declaración-o-expresión>* )`

18.3. Creación de instancias

Para crear una instancia de una clase se utiliza el bloque `new`:
`(new <nombre-clase> <parámetro-de-inicialización>* )`

También podría utilizarse en la forma:
`(new (class ...) <parámetro-de-inicialización>* )`, pero sería poco legible y sólo se prodría crear una instancia de esa clase. Aunque si sólo se creará una instancia de la clase, esta forma es conveniente, igual que usar `lambda` para funciones que sólo se invocarán una vez.

La forma de `<parámetro-de-inicialización>` se describe en la sección 18.5.

18.4. Métodos

18.4.1. Definición e Invocación de Métodos

Dentro del cuerpo de una clase, en su sección de definiciones, se definen los métodos de la clase con el bloque:
`(define/public (<nombre-método> <parámetros>* ) <exp-cuerpo>* )`

Este bloque `define/public` es una forma abreviada de:

```
(begin
  (public <nombre-método>)
  (define <nombre-método> <exp-lambda_con_PARÁMETROS_y_EXP-CUERPO> )
  )
```

El bloque `public` sirve para indicar todos los identificadores de las funciones que serán *públicas* (igual que el bloque `provide` para hacer *visibles o públicos* ciertos identificadores de los módulos).

Para invocar un método de un objeto, se utiliza el bloque `send`:
`(send <instancia> <nombre-método> <parámetros>* )`

18.4.2. Sustitución de métodos

En algunos casos, una clase hija debe redefinir métodos de clase. Esto se hace con el bloque `define/override`:
`(define/override (<nombre-método> <parámetros>* ) <exp-cuerpo>* )`

Este bloque `define/override` es una forma abreviada de:

```
(begin
  (override <nombre-método>)
  (define <nombre-método> <exp-lambda_con_PARÁMETROS_y_EXP-CUERPO> )
  )
```

El bloque `override` sirve para indicar todos los identificadores de las funciones que serán *sobreescritos* en esta clase.

18.4.3. Métodos no sustituíbles

En algunos casos, una clase debe (re)definir métodos de clase que no puedan ser *sobreescritos* por sus descendientes. Esto se hace con los bloques `define/public-final` y `define/override-final` (dependiendo de si es un método nuevo en la clase actual o uno heredado):

```
(define/public-final (<nombre-método> <parámetros>* ) <exp-cuerpo>* )
(define/override-final (<nombre-método> <parámetros>* ) <exp-cuerpo>* )
```

Ambos bloques `define/public-final` y `define/override-final` respectivamente son formas abreviadas de:

```
(begin
  (public-final <nombre-método>)
  (define <nombre-método> <exp-lambda_con_PARÁMETROS_y_EXP-CUERPO> )
  )
```

y

```
(begin
  (override-final <nombre-método>)
  (define <nombre-método> <exp-lambda_con_PARÁMETROS_y_EXP-CUERPO> )
  )
```

Los bloques `public-final` y `override-final` sirven para indicar todos los identificadores de funciones que no podrán ser *sobreescritos* en las clases descendientes de esta.

18.5. Parámetros de inicialización

Para indicar que la inicialización de una instancia de una clase requiere parámetros, se utiliza el bloque `init` dentro de la definición de la clase:

```
( class <superclase>
    <declaración-o-expresión>*
    (init <parámetro-formal>* )
    <declaración-o-expresión>*
)
```

donde:

```
<parámetro-formal> ::= <identificador> | [<identificador> <exp-valor-por-defecto>]
```

Para crear una instancia y pasarle los parámetros correspondientes, se utiliza el mismo bloque `new`, pero pasando los parámetros con sus respectivos nombres de parámetro formal:

```
(new <nombre-clase> <parámetro-real>* )
```

donde:

```
<parámetro-real> ::= [<identificador> <valor>]
```

Supongamos que una clase `A` define sus parámetros de inicialización (tiene su `init`), y una clase `B` no define parámetros de inicialización (no tiene `init`). Entonces la instanciación de `B` puede llevar los parámetros indicados en `A` y estos serán pasados al inicializador de `A` cuando en `B` se encuentre la llamada a `(super-new)`.

Supongamos que `A` define sus parámetros de inicialización y que su clase derivada `B` también define los suyos, diferentes de los de `A`. Entonces, en la inicialización de `B` debe hacerse una invocación a `super-new` con los parámetros correspondientes para la inicialización de `A`, con el bloque:
`(super-new <parámetro-real>* )`

18.6. Funciones que operan sobre clases/interfaces/objetos

- `object?` toma un valor y verifica si es un objeto o no.
- `class?` toma un valor y verifica si es una clase o no.
- `interface?` toma un valor y verifica si es una clase o no.
- `object=?` toma dos objetos y verifica si son el mismo.
- `class->interface` toma una clase y devuelve la interface implícitamente definida por la clase.
- `object-interface` toma un objeto y devuelve la interface implícitamente declara por la clase del objeto.
- `is-a?` toma un valor y una clase o interface, y verifica si el valor es una instancia de la clase, o si el valor es una instacia de alguna clase que implemente la interface proporcionada.
- `subclass?` toma un valor y una clase, y verifica si el valor es una clase derivada de la clase proporcionada.
- `implementation?` toma un valor y una interface y verifica si el valor es una clase que implementa la interface proporcionada.
- `implementation-extension?` toma un valor y una interface y verifica si el valor es una interface que extiende a la interface proporcionada.
- `method-in-interface?` toma un símbolo y una interface y verifica si la interface (o alguna de sus ancestros) contiene un miembro con el mismo nombre que el símbolo.
- `interface->method-names` toma una interface y devuelve una lista de símbolos que indican los nombres de los miembros de la interface y de sus ancestros.
- `object-method-arity-includes?` toma un objeto, un símbolo y un entero positivo y verifica si el objeto en cuestión tiene un método llamado como el símbolo y que además acepte ese número de parámetros.

18.7. Ejemplos

Este es un ejemplo sobre definición de clases y métodos:

```
#lang racket
;cola.rkt
;;Definición de una clase Cola

(define cola%
  (class object%

    (super-new) ;;Siempre debe inicializarse la superclase

    (define elementos '() ) ;; Este valor es "privado", dentro de la clase.

    (define/public (tamaño)
      (length elementos)
      )
    (define/public (meter nuevo-elemento)
      (set! elementos (cons nuevo-elemento elementos))
      )
    (define/public (sacar)
      (if (empty? elementos)
        (error "¡Intento de extraer un elemento de una Cola vacía!")
        (let ([e (last elementos)] ;;'last' devuelve el último elemento de
            una lista
              ;;'take' devuelve una lista con los primeros elementos de una
                  lista
              [lista (take elementos (sub1 (length elementos)))])
          (set! elementos lista)
          e
          )
        )
      )
    )
  )
```

Ejemplos de interacción con la clase `cola%`:

```
> (define colita (new cola%))
> colita
#(struct:object:cola% ...)
> (send colita tamaño)
0
> (send colita sacar)
. . ¡Intento de extraer un elemento de una Cola vacía!
> (send colita meter 1234)
> (send colita tamaño)
1
> (for-each (lambda (e) (send colita meter e)) '("1" "2" "3" "4"))
> (send colita tamaño)
```

```
5
> (send colita sacar)
1234
> (send colita tamaño)
4
> (send colita sacar)
"1"
```

Ejemplo para aclarar el significado de `define/public`:

```
#lang racket
;pila.rkt
;;Definición de una clase Pila

(define pila%
  (class object%

    (super-new)

    (define elementos '() )

    (public tamaño) ;;'public' y 'define' separados
    (define (tamaño)
      (length elementos)
      )

    (public meter) ;;'public' y 'define' con 'lambda' separados
    (define meter
      (lambda (nuevo-elemento)
        (set! elementos (cons nuevo-elemento elementos))
        )
      )

    (define/public (sacar) ;;'public' y 'define' juntos
      (if (empty? elementos)
        (error "¡Intento de extraer un elemento de una Pila vacía!")
        (let ([e (first elementos)]
              [lista (rest elementos)])
          (set! elementos lista)
          e
          )
        )
      )

    )
  )
```

Ejemplos de interacción con la clase `pila%`:

```
> (define pilita (new pila%))
> (new pila%)
#(struct:object:pila% ...)
```

```
> (send pilita tamaño)
0
> (send pilita meter sqrt)
> (send pilita tamaño)
1
> ((send pilita sacar) 81)
9
> (send pilita tamaño)
0
> (for-each (lambda (e) (send pilita meter e)) '(cos sqrt 3 "hola"))
> ((lambda (pila)
     (define (sacar-todo p)
       (if (positive? (send p tamaño))
           (begin
             (printf "Elemento: ~a\n" (send p sacar))
             (sacar-todo p)
             )
           (void)
           )
       )
     (sacar-todo pila)
     )
   pilita
   )
Elemento: hola
Elemento: 3
Elemento: sqrt
Elemento: cos

-
```

Parte IV

Interfaces Gráficas de Usuario

19 Introducción a las interfaces gráficas de usuario con Racket

A continuación se describen algunos tópicos básicos sobre interfáces gráficas de usuario en Racket.

Para poder usar interfaces gráficas de usuario en Racket, es necesario agregar el módulo `gui`, con la línea:
`(require racket/gui)`

En ese módulo está definida la jerarquía de clases, y las clases, del modelo de interfaces de Racket.

19.1. Hola Mundo

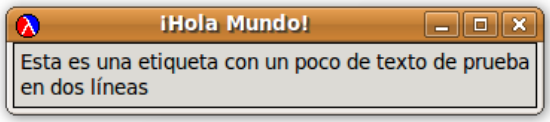


Figura 19.1: hola-mundo.rkt

```
1  #lang racket
2  ;hola-mundo.rkt
3
4  ;invocar al módulo gui
5  (require racket/gui)
6
7  ;Crear una ventanta
8  (define ventana (new frame% [label "¡Hola Mundo!"]))
9
10 ;Crear y ponerle un objeto 'message%' a la ventana
11 (define mensaje (new message% [parent ventana]
12                        [label "Esta es una etiqueta con un poco de texto de
                              prueba\nen dos líneas"]))
13
14 ;Mostrar la ventana al usuario
15 (send ventana show #t)
```

19.2. Ejecución y compilación

Para ejecutar el programa anterior y cualquier otro programa Racket desde línea de comandos que incluya interfaces gráficas de usuario, debe usarse el comando `gracket` en lugar del `racket` tradicional:

```
$ gracket hola-mundo.rkt
```

Y para compilar, hay que incluir la opción `--gui` en la línea de compilación:

```
$ raco exe -o ejecutable --gui hola-mundo.rkt
```

19.3. Introducción a los eventos

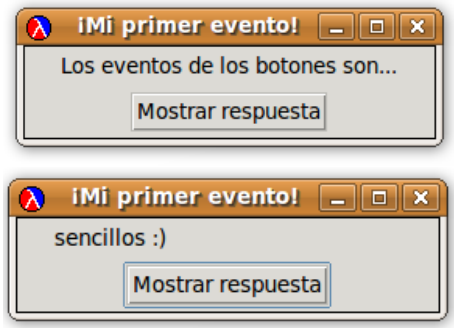


Figura 19.2: eventos-1.rkt

Existen varios mecanismos para manejar eventos. Entre ellos, algunos objetos, permiten indicar un procedimiento de *callback* que se ejecutará cuando el evento suceda.

```
1  #lang racket
2  ;eventos-1.rkt
3
4  (require racket/gui)
5
6  (define ventana (new frame% [label "¡Mi primer evento!"]))
7
8  (define mensaje (new message% [parent ventana]
9                       [label "Los eventos de los botones son..."]))
10
11 (new button% [parent ventana]
12      [label "Mostrar respuesta"]
13      ; Función a invocar cuando se presione el botón.
14      ; Debe ser una función binaria, recibe una referencia al botón,
15      ; y una instancia 'control-event%'.
16      [callback (lambda (botón evento)
17                  ;Los objetos 'message%' tienen un método 'set-label'
```

```
18                   (send mensaje set-label "sencillos :)"))
19                 ]
20       )
21
22  (send ventana show #t)
```

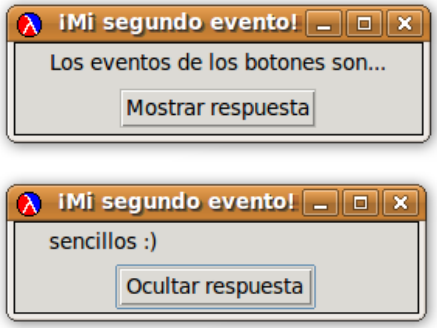


Figura 19.3: eventos-2.rkt

```
1  #lang racket
2  ;eventos-2.rkt
3
4  (require racket/gui)
5
6  (define ventana (new frame% [label "¡Mi segundo evento!"]))
7
8  (define mensaje (new message% [parent ventana]
9                       [label "Los eventos de los botones son..."]))
10
11 ; ¿Se está mostrando la respuesta?
12 (define respuesta-mostrada #f)
13
14 (define mi-botón
15   (new button% [parent ventana]
16        [label "Mostrar respuesta"]
17        [callback (lambda (botón evento)
18                    (if respuesta-mostrada
19                        (begin
20                          (send mensaje set-label "Los eventos de los
                                botones son...")
21                          (send mi-botón set-label "Mostrar respuesta")
22                          (set! respuesta-mostrada #f)
23                          )
24                        (begin
25                          (send mensaje set-label "sencillos :)")
26                          (send mi-botón set-label "Ocultar respuesta")
27                          (set! respuesta-mostrada #t)
28                          )
29                        )
```

```
30                      )
31                    ]
32          )
33    )
34
35  (send ventana show #t)
```

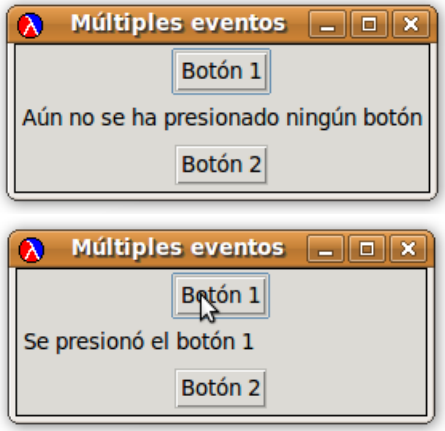


Figura 19.4: eventos-3.rkt

Una misma función de *callback* se puede usar para procesar varios eventos:

```
1  #lang racket
2  ;eventos-3.rkt
3
4  (require racket/gui)
5
6  (define ventana (new frame% [label "Múltiples eventos"]))
7
8  (define (función-manejadora b c)
9    (if (object=? b botón1)
10       (send mensaje set-label "Se presionó el botón 1")
11       (send mensaje set-label "Se presionó el botón 2")
12       )
13   )
14 (define botón1 (new button%
15                    [parent ventana]
16                    [label "Botón 1"]
17                    [callback función-manejadora]
18                    ))
19
20 (define mensaje (new message% [parent ventana]
21                     [label "Aún no se ha presionado ningún botón"]))
22
23 (define botón2 (new button%
```

```
24                      [parent ventana]
25                      [label "Botón 2"]
26                      [callback función-manejadora]
27                      ))
28
29  (send ventana show #t)
```

19.4. Ventanas de diálogo

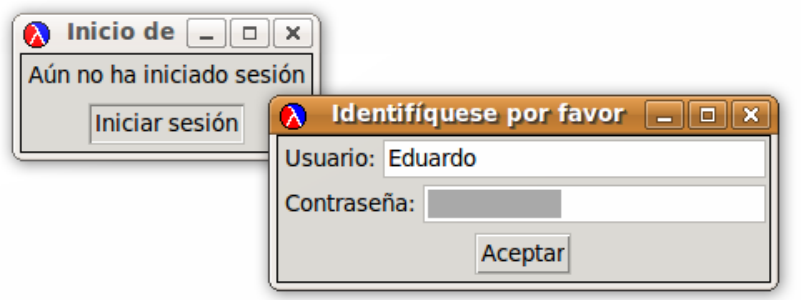


Figura 19.5: dialogo.rkt

Se pueden lanzar ventanas de diálogo, de manera muy sencilla, con la clase `dialog%`:

```
1   #lang racket
2   ;dialogo.rkt
3
4   (require racket/gui)
5
6   ;Ventana principal
7   (define ventana (new frame% [label "Inicio de sesión"]))
8
9   (define datos (new message%
10                     [label "Aún no ha iniciado sesión"]
11                     [parent ventana]
12                     [auto-resize #t]
13                     ))
14
15  (define lanzar
16    (new button%
17         [parent ventana]
18         [label "Iniciar sesión"]
19         [callback
20          (lambda (b c)
21            (send ventana-de-diálogo show #t)
22            (send datos set-label
23                  (string-append "Usuario: '"
```

```
24                                  (send txt-usuario get-value)
25                                  "', Contraseña: '"
26                                  (send txt-contraseña get-value)
27                                  "'"
28                                  )
29                  )
30              )]
31          ))
32
33  ;La otra ventana, de diálogo
34  (define ventana-de-diálogo (new dialog% [label "Identifíquese por favor"]))
35
36  (define txt-usuario (new text-field%
37                           [label "Usuario:"]
38                           [parent ventana-de-diálogo]
39                           ))
40
41  (define txt-contraseña (new text-field%
42                           [label "Contraseña:"]
43                           [parent ventana-de-diálogo]
44                           [style (list 'single 'password)]
45                           ))
46  (new button%
47       [parent ventana-de-diálogo]
48       [label "Aceptar"]
49       [callback (lambda (b c) (send ventana-de-diálogo show #f))]
50       )
51
52
53  (send ventana show #t)
```

19.5. Eventos de cuadros de texto

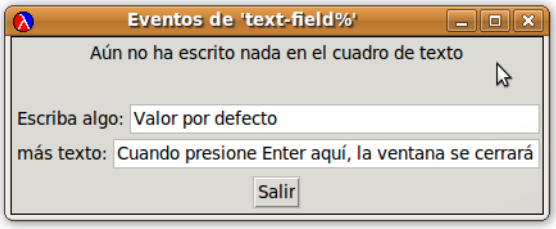


Figura 19.6: text-field.rkt

```
1  #lang racket
2  ;text-field.rkt
3
```

```
4  (require racket/gui)
5
6  ;Ventana principal
7  (define ventana (new frame% [label "Eventos de 'text-field%'"]))
8
9  (define mensaje (new message%
10                       [label "Aún no ha escrito nada en el cuadro de texto"]
11                       [parent ventana]
12                       ))
13 (define copia-contenido (new message%
14                              [label ""]
15                              [parent ventana]
16                              [auto-resize #t]))
17
18 (define texto
19   (new text-field%
20        [label "Escriba algo:"]
21        [parent ventana]
22        [init-value "Valor por defecto"]
23        [callback
24         (lambda (obj control)
25           (case (send control get-event-type)
26             ['text-field
27              (send mensaje set-label "Evento 'text-field")
28              (send copia-contenido set-label (send texto get-value))
29              ]
30             ['text-field-enter
31              (send mensaje set-label "Evento 'text-field-enter")
32              ]
33            )
34           )]
35        ))
36
37 (define otro-texto
38   (new text-field%
39        [label "más texto:"]
40        [parent ventana]
41        [init-value "Cuando presione Enter aquí, la ventana se cerrará"]
42        [callback
43         (lambda (o c)
44           (if (equal? 'text-field-enter (send c get-event-type))
45               ;Esto dispara la función de 'callback' de 'btn-salir':
46               (send btn-salir command (new control-event% [event-type '
                     button]))
47               (void)
48               )
49           )]
50        ))
51
52 (define btn-salir (new button%
53                        [label "Salir"]
```

```
54                          [parent ventana]
55                          [callback (lambda (b c)
56                                      (send ventana show #f)
57                                      )]))
58
59  (send ventana show #t)
```

19.6. Páneles

Figura 19.7: páneles.rkt

```
1   #lang racket
2   ;páneles.rkt
3
4   (require racket/gui)
5
6   (define ventana (new frame% [label "Páneles horizontales"]))
7
8   (define mensaje (new message%
9                        [parent ventana]
10                       [label "Páneles horizontales"]
11                       [auto-resize #t]
12                       ))
13
14  ;(define panel (new horizontal-pane% [parent ventana]))
15  (define panel (new horizontal-panel% [parent ventana]))
16  #|
17  La diferencia entre 'pane%' y 'panel%' es que
18  'pane%' sólo sirve para administrar la distribución
19  de los demás controles, no puede ser ocultado o deshabilitado,
```

```
20 ni soporta manejo de eventos.
21 |#
22
23 (new button% [parent panel]
24      [label "Botón Izquierdo"]
25      [callback (lambda (button event)
26                  (send mensaje set-label "Clic del botón izquierdo"))])
27 (new button% [parent panel]
28      [label "Botón Derecho"]
29      [callback (lambda (button event)
30                  (send mensaje set-label "Clic del botón derecho"))])
31
32
33 (send ventana show #t)
```

20 Uso de los diversos controles de Racket

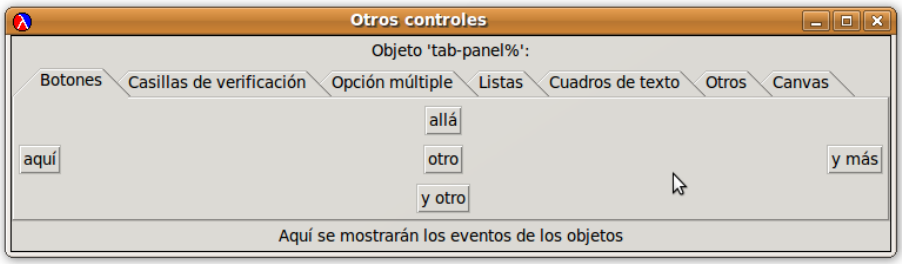


Figura 20.1: controles.rkt, tab 1

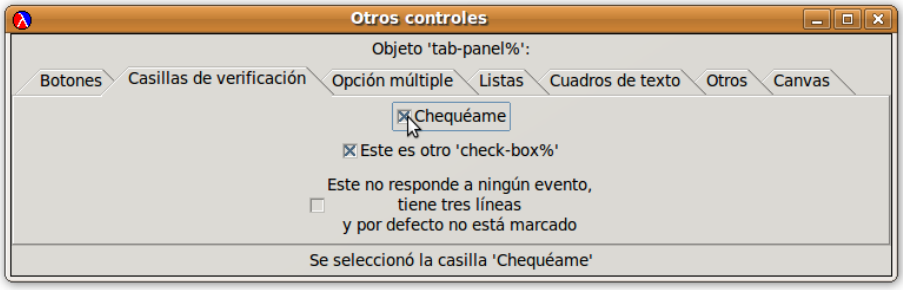


Figura 20.2: controles.rkt, tab 2

```
1  #lang racket
2  ;controles.rkt
3
4  (require racket/gui)
5
6  (define ventana (new frame% [label "Otros controles"]))
7
8  ;;Objetos de la ventana principal:
       ------------------------------------------
9  (define texto (new message%
10                     [parent ventana]
11                     [label "Objeto 'tab-panel%':"]
```

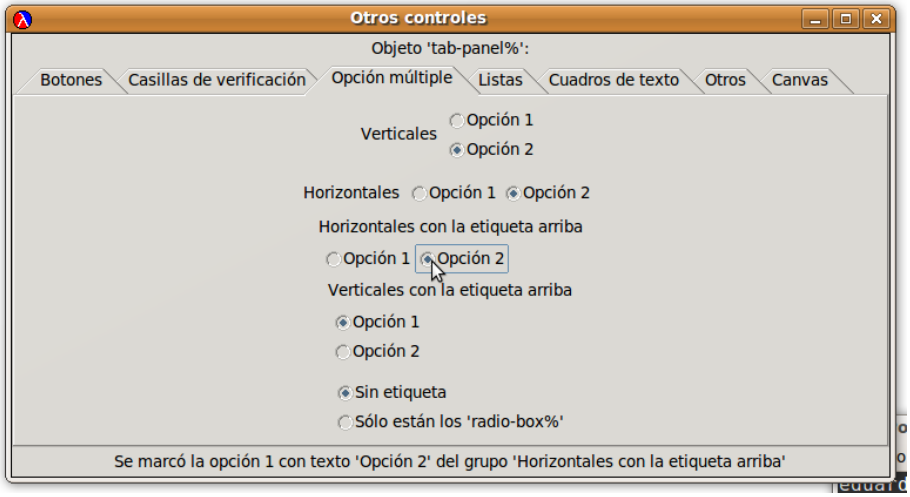


Figura 20.3: controles.rkt, tab 3

```
12                    ))
13  (define opciones (list "Botones" "Casillas de verificación" "Opción
        múltiple" "Listas" "Cuadros de texto" "Otros" "Canvas"))
14  (define tab (new tab-panel%
15                   [parent ventana]
16                   [choices opciones]
17                   [callback
18                    (lambda (t c) ;una instancia de 'tab-panel%' y de '
                          control-event%'
19                      (let* ([ix (send tab get-selection)]
20                             [ix-str (number->string ix)]
21                             [nombre-ficha (send tab get-item-label ix)])
22                        (send mensaje set-label (string-append "Se seleccionó
                               la ficha " ix-str " con nombre '" nombre-ficha
                              "'"))
23                        (send tab delete-child (first (send tab get-children)
                              ))
24                        (send tab add-child (list-ref lista-páneles ix))
25                        )
26                      )]
27                   ))
28  (define mensaje (new message%
29                       [parent ventana]
30                       [label "Aquí se mostrarán los eventos de los objetos"]
31                       [auto-resize #t]
32                       ))
33
34
35  ;;Objetos páneles:
        ------------------------------------------------------------
```

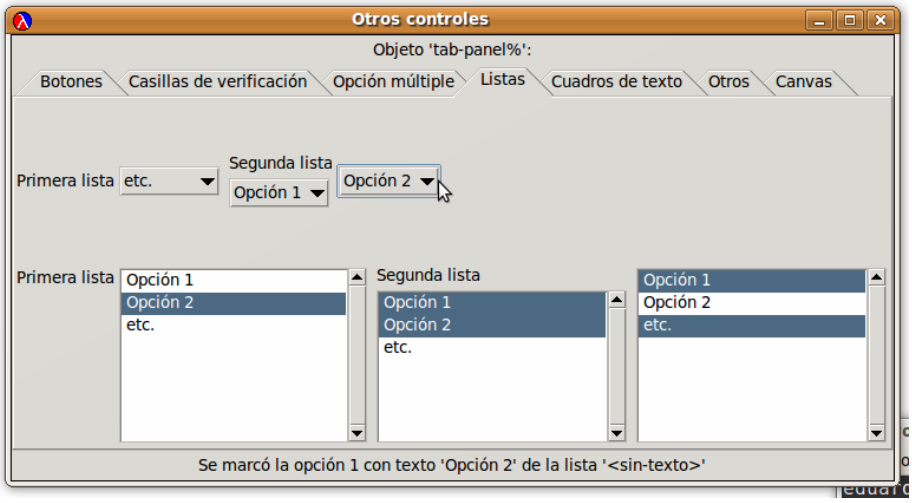


Figura 20.4: controles.rkt, tab 4

```
36 (define panel0 (new vertical-panel% [parent tab]))
37 (define panel1 (new vertical-panel% [parent tab][style '(deleted)]))
38 (define panel2 (new vertical-panel% [parent tab][style '(deleted)]))
39 (define panel3 (new vertical-panel% [parent tab][style '(deleted)]))
40 (define panel4 (new vertical-panel% [parent tab][style '(deleted)]))
41 (define panel5 (new vertical-panel% [parent tab][style '(deleted)]))
42 (define panel6 (new vertical-panel% [parent tab][style '(deleted)]))
43
44 (define lista-páneles (list panel0 panel1 panel2 panel3 panel4 panel5
       panel6))
45
46 ;;Panel 0 - Botones
       ------------------------------------------------------------
47 (define panel-botones (new horizontal-pane% [parent panel0]))
48 (define (función-botón b c)
49   (send mensaje set-label (string-append "Clic en el botón '" (send b get-
         label) "'"))
50   )
51 (define botón1 (new button%
52                     [parent panel-botones]
53                     [label "aquí"]
54                     [callback función-botón]))
55 (define panel-botones2 (new vertical-pane% [parent panel-botones]))
56 (define botón2 (new button%
57                     [parent panel-botones2]
58                     [label "allá"]
59                     [callback función-botón]))
60 (define botón3 (new button%
61                     [parent panel-botones2]
62                     [label "otro"]
```

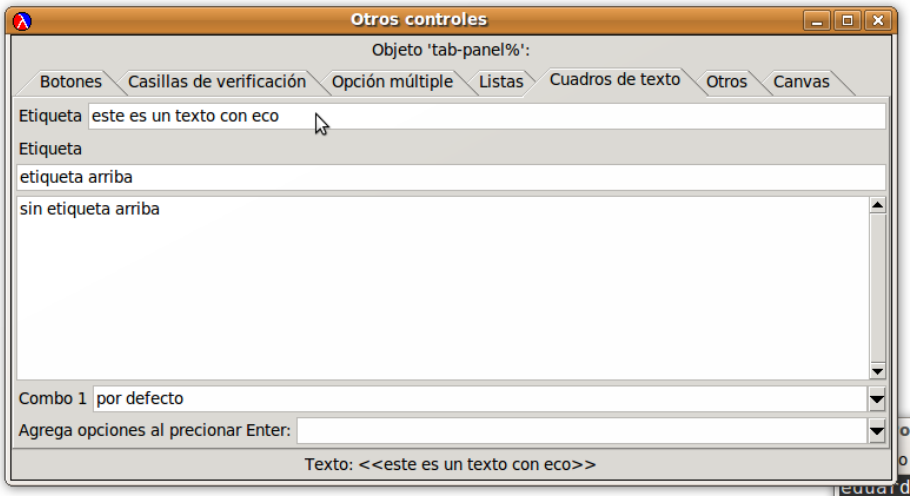


Figura 20.5: controles.rkt, tab 5

```
63                      [callback función-botón]))
64  (define botón4 (new button%
65                      [parent panel-botones2]
66                      [label "y otro"]
67                      [callback función-botón]))
68  (define botón5 (new button%
69                      [parent panel-botones]
70                      [label "y más"]
71                      [callback función-botón]))
72
73  ;;Panel 1 - Casillas de verificación
      ---------------------------------------
74  (define (función-chequeo c e) ;una instancia 'check-box%' y de 'control-
      event%'
75    (send mensaje set-label
76          (string-append "Se "
77                         (if (send c get-value) "seleccionó" "deseleccionó")
78                         " la casilla '" (send c get-label) "'"))
79    )
80  (define chequeo1 (new check-box%
81                        [label "Chequéame"]
82                        [parent panel1]
83                        [value #f]
84                        [callback función-chequeo]))
85  (define chequeo2 (new check-box%
86                        [label "Este es otro 'check-box%'"]
87                        [parent panel1]
88                        [value #t]
89                        [callback función-chequeo]))
90  (define chequeo3 (new check-box%
```

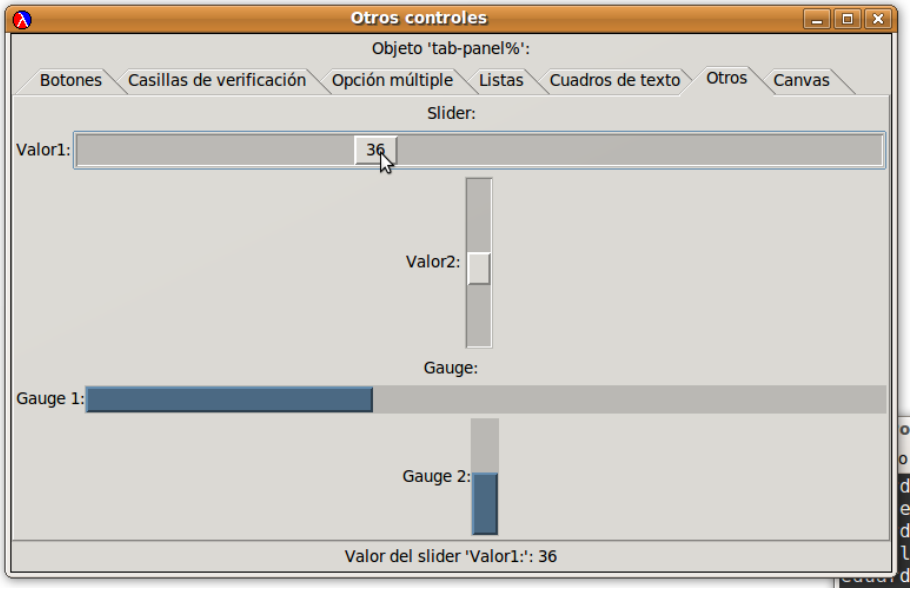


Figura 20.6: controles.rkt, tab 6

```
91                          [label "Este no responde a ningún evento,\ntiene tres
                                líneas\ny por defecto no está marcado"]
92                          [parent panel1]
93                          ))
94
95  ;;Panel 2 - Opción múltiple ------------------------------------------------
96  (define (función-opción-múltiple r c) ;instancia 'radio-box%' y 'control-
        event%'
97    (send mensaje set-label
98          (string-append "Se marcó la opción "
99                         (number->string (send r get-selection))
100                        " con texto '"
101                        (send r get-item-label (send r get-selection))
102                        "' del grupo '"
103                        (let ([t (send r get-label)])
104                          (if t t "<sin-texto>"))
105                        "'"
106                        ))
107   )
108 ;(send panel2 set-orientation #t) ;orientación horizontal
109 (define panel-radio1 (new radio-box%
110                           [label "Verticales"]
111                           [parent panel2]
112                           [choices (list "Opción 1" "Opción 2")]
113                           [callback función-opción-múltiple]
114                           ))
115 (define panel-radio2 (new radio-box%
```

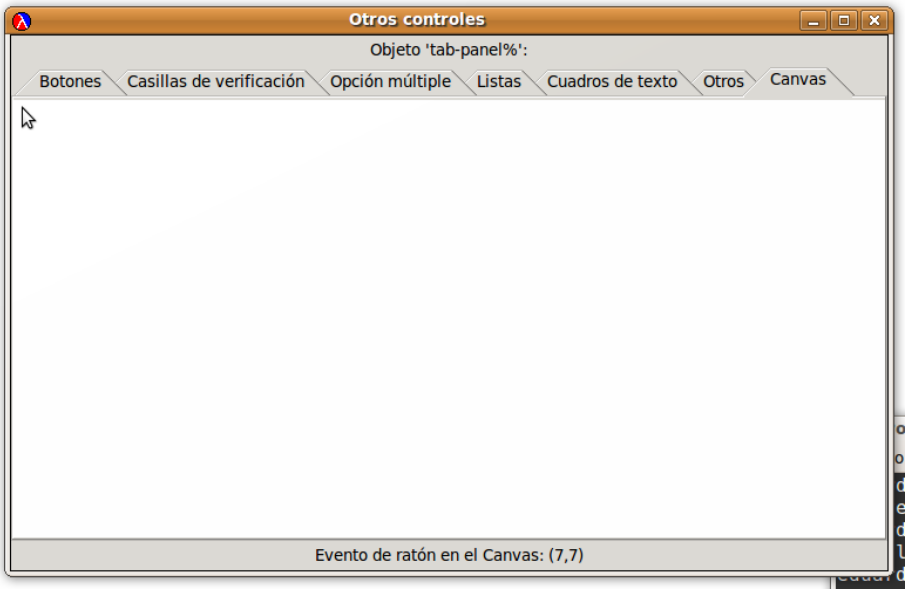


Figura 20.7: controles.rkt, tab 7

```
116                          [label "Horizontales"]
117                          [parent panel2]
118                          [choices (list "Opción 1" "Opción 2")]
119                          [callback función-opción-múltiple]
120                          [style (list 'horizontal)]
121                          [selection 1]
122                          ))
123 (define panel-radio3 (new radio-box%
124                          [label "Horizontales con la etiqueta arriba"]
125                          [parent panel2]
126                          [choices (list "Opción 1" "Opción 2")]
127                          [callback función-opción-múltiple]
128                          [style (list 'horizontal 'vertical-label)]
129                          ))
130 (define panel-radio4 (new radio-box%
131                          [label "Verticales con la etiqueta arriba"]
132                          [parent panel2]
133                          [choices (list "Opción 1" "Opción 2")]
134                          [callback función-opción-múltiple]
135                          [style (list 'vertical 'vertical-label)]
136                          ))
137 (define panel-radio5 (new radio-box%
138                          [label #f]
139                          [parent panel2]
140                          [choices (list "Sin etiqueta" "Sólo están los '
                                radio-box%'")]
141                          [callback función-opción-múltiple]
```

```
                          ))

;;Panel 3 - Listas ------------------------------------------------------
(define (función-lista r c) ;instancia de lista y 'control-event%'
  (send mensaje set-label
        (string-append "Se marcó la opción "
                       (number->string (send r get-selection))
                       " con texto '"
                       (send r get-string-selection)
                       "' de la lista '"
                       (let ([t (send r get-label)])
                         (if t t "<sin-texto>"))
                       "'"
                       ))
  )
(define panel-lista1 (new horizontal-pane% [parent panel3]))
(define elección1 (new choice%
                       [label "Primera lista"]
                       [choices (list "Opción 1" "Opción 2" "etc.")]
                       [parent panel-lista1]
                       [callback función-lista]
                       [selection 2]
                       ))
(define elección2 (new choice%
                       [label "Segunda lista"]
                       [choices (list "Opción 1" "Opción 2" "etc.")]
                       [parent panel-lista1]
                       [callback función-lista]
                       [style (list 'vertical-label)]
                       ))
(define elección3 (new choice%
                       [label #f]
                       [choices (list "Opción 1" "Opción 2" "etc.")]
                       [parent panel-lista1]
                       [callback función-lista]
                       ))
(define panel-lista2 (new horizontal-pane% [parent panel3]))
(define lista1 (new list-box%
                    [label "Primera lista"]
                    [choices (list "Opción 1" "Opción 2" "etc.")]
                    [parent panel-lista2]
                    [callback función-lista]
                    [selection 2]
                    ))
(define lista2 (new list-box%
                    [label "Segunda lista"]
                    [choices (list "Opción 1" "Opción 2" "etc.")]
                    [parent panel-lista2]
                    [callback función-lista]
                    [style (list 'multiple 'vertical-label)]
                    ))
```

```
193 (define lista3 (new list-box%
194                     [label #f]
195                     [choices (list "Opción 1" "Opción 2" "etc.")]
196                     [parent panel-lista2]
197                     [callback función-lista]
198                     [style (list 'extended 'vertical-label)]
199                     ))
200 ;ver adicionalmente, los métodos 'get-selections' y 'is-selected?' de 'list
        -box%'
201
202 ;;Panel 4 - Cuadros de texto ------------------------------------------
203 (define (función-texto t c)
204   (send mensaje set-label (string-append "Texto: <<" (send t get-value)
          ">>"))
205   )
206 (define texto1 (new text-field%
207                     [label "Etiqueta"]
208                     [parent panel4]
209                     [init-value "escriba algo"]
210                     [callback función-texto]
211                     ))
212 (define texto2 (new text-field%
213                     [label "Etiqueta"]
214                     [parent panel4]
215                     [init-value "etiqueta arriba"]
216                     [style (list 'vertical-label 'single)]
217                     [callback función-texto]
218                     ))
219 (define texto3 (new text-field%
220                     [label #f]
221                     [parent panel4]
222                     [init-value "sin etiqueta arriba"]
223                     [style (list 'multiple)]
224                     [callback función-texto]
225                     ))
226 (define combo1 (new combo-field%
227                     [label "Combo 1"]
228                     [parent panel4]
229                     [choices (list "Opción 1" "Opción 2" "etc.")]
230                     [init-value "por defecto"]
231                     [callback (lambda (c e)
232                                 (send mensaje set-label
233                                       (string-append "Evento del 'combo-
                                              field%' "
234                                                      (send c get-label)
235                                                      )))]
236                     ))
237 (define combo2 (new combo-field%
238                     [label "Agrega opciones al precionar Enter:"]
239                     [parent panel4]
240                     ;[choices (list "Opción 1" "Opción 2" "etc.")]
```

```
241                      [choices null]
242                      [callback
243                       (lambda (c e)
244                         (when (equal? (send e get-event-type) 'text-field-
                                enter)
245                           (begin
246                             (send c append (send c get-value))
247                             (send mensaje set-label
248                                   (string-append "Texto agregado: '"
249                                                  (send c get-value)
250                                                  "'"))
251                             )
252                           )
253                         )]
254                      ))
255 ;;Panel 5 - Otros ------------------------------------------------------
256 (define msg-otros1 (new message%
257                          [label "Slider:"]
258                          [parent panel5]))
259
260 (define (evento-slider s e)
261   (send mensaje set-label
262         (string-append "Valor del slider '"
263                        (send s get-label)
264                        "': "
265                        (number->string (send s get-value))))
266   (if (object=? s slider1)
267       (send gauge1 set-value (send s get-value))
268       (send gauge2 set-value (send s get-value))
269       )
270   )
271 (define slider1 (new slider%
272                      [label "Valor1:"]
273                      [parent panel5]
274                      [min-value 0]
275                      [max-value 100]
276                      [init-value 30]
277                      [callback evento-slider]
278                      ))
279 (define slider2 (new slider%
280                      [label "Valor2:"]
281                      [parent panel5]
282                      [min-value 0]
283                      [max-value 100]
284                      [style (list 'vertical 'plain)]
285                      [callback evento-slider]
286                      ))
287
288 (define msg-otros2 (new message%
289                          [label "Gauge:"]
290                          [parent panel5]))
```

```
(define gauge1 (new gauge%
                    [label "Gauge 1:"]
                    [parent panel5]
                    [range 100]
                    ))
(send gauge1 set-value (send slider1 get-value))
(define gauge2 (new gauge%
                    [label "Gauge 2:"]
                    [parent panel5]
                    [style '(vertical)]
                    [range 100]
                    ))
(send gauge2 set-value (send slider2 get-value))

;;Panel 6 - Canvas ------------------------------------------------------
(define canvas-hijo%
  (class canvas%
    (define/override (on-event evento)
      (send mensaje set-label
            (string-append "Evento de ratón en el Canvas: ("
                           (number->string (send evento get-x))
                           ","
                           (number->string (send evento get-y))
                           ")"
                           )
            ))
    (define/override (on-char evento)
      (send mensaje set-label
            (string-append "Evento de teclado en el Canvas: "
                           (let ([t (send evento get-key-code)])
                             (if (char? t) (string t) (symbol->string t))
                             )
                           )))
    (super-new)
    ))

(define c (new canvas-hijo%
               [parent panel6]
               ))

;;Finalmente mostrar la ventana: ----------------------------------------
(send ventana show #t)
```

21 Dibujo con Lienzos

21.1. Dibujo en un `canvas%`

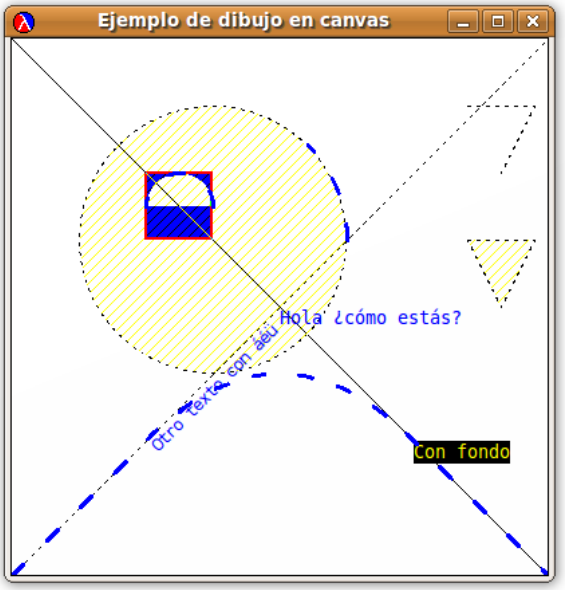


Figura 21.1: canvas1.rkt

```
1  #lang racket
2  ;canvas1.rkt
3  (require racket/gui)
4
5  (define ventana (new frame%
6                       [label "Ejemplo de dibujo en canvas"]
7                       [width 400]
8                       [height 400]))
9  ; Objeto 'canvas%'
10 (define canvas (new canvas%
11                     [parent ventana]
12                     [paint-callback (lambda (c dc) ;instancia de 'canvas%'
                           y 'dc<%>'
13                                     (dibujar dc)
14                                     )]
```

```
15                      ))
16  ; Colores:
17  (define negro (make-object color% 0 0 0)) ; RGB
18  (define rojo (make-object color% 255 0 0))
19  (define azul (make-object color% 0 0 255))
20  (define amarillo (make-object color% 255 255 0))
21
22  ; Lápices y brochas
23  (define lápiz-negro-punteado (make-object pen% negro 1 'dot)) ;color grueso
        estilo
24  (define lápiz-rojo-sólido (make-object pen% rojo 2 'solid))
25  (define lápiz-azul-líneas (make-object pen% azul 3 'long-dash))
26  (define lápiz-negro-invertido (make-object pen% negro 1 'xor))
27  (define lápiz-transparente (make-object pen% negro 1 'transparent))
28
29  (define brocha-negra-sólida (make-object brush% negro 'solid)) ; color
        estilo
30  (define brocha-azul-invertida (make-object brush% azul 'xor))
31  (define brocha-amarilla-rallada (make-object brush% amarillo 'bdiagonal-
        hatch))
32  (define brocha-transparente (make-object brush% negro 'transparent))
33
34  ; Función de dibujo
35  (define (dibujar dc) ; recibe una instancia del 'drawing context'
36    (send dc set-pen lápiz-negro-punteado)
37    (send dc set-brush brocha-amarilla-rallada)
38    (send dc draw-ellipse 50 50 200 200) ;x y ancho alto
39
40    (send dc set-pen lápiz-rojo-sólido)
41    (send dc set-brush brocha-azul-invertida)
42    (send dc draw-rectangle 100 100 50 50) ; x y ancho alto
43
44    (send dc set-pen lápiz-azul-líneas)
45    (send dc draw-arc 100 100 50 50 0.0 pi); x y ancho alto rad-inicio rad-
          fin
46
47    (send dc set-brush brocha-transparente)
48    (send dc draw-arc 50 50 200 200 0.0 (/ pi 4))
49
50    (send dc set-pen lápiz-negro-invertido)
51    (send dc draw-line 0 0 400 400) ;x1 y1 x2 y2
52    (send dc set-pen lápiz-negro-punteado)
53    (send dc draw-line 0 400 400 0)
54
55    (send dc set-text-background negro) ; color
56    (send dc set-text-foreground azul) ; color
57    (send dc draw-text "Hola ¿cómo estás?" 200 200) ; texto x y
58    (send dc draw-text "Otro texto con áéü" 100 300 #f 0 (/ pi 4)) ;texto x y
          combinar? despl ángulo
59    (send dc set-text-mode 'solid) ;el otro es 'transparent
60    (send dc set-text-foreground amarillo)
```

```
61    (send dc draw-text "Con fondo" 300 300)
62
63    (send dc set-pen lápiz-azul-líneas)
64    (send dc draw-spline 0 400 200 200 400 400)
65
66    (send dc set-pen lápiz-negro-punteado)
67    (send dc set-brush brocha-amarilla-rallada)
68    (let ([puntos (list (make-object point% 0 0)
69                        (make-object point% 50 0)
70                        (make-object point% 25 50))])
71      (send dc draw-lines puntos 340 50) ;puntos [x y]
72      (send dc draw-polygon puntos 340 150) ; puntos [x y estilo]
73      )
74
75    )
76
77  (send ventana show #t)
```

21.2. Interacción avanzada con canvas%

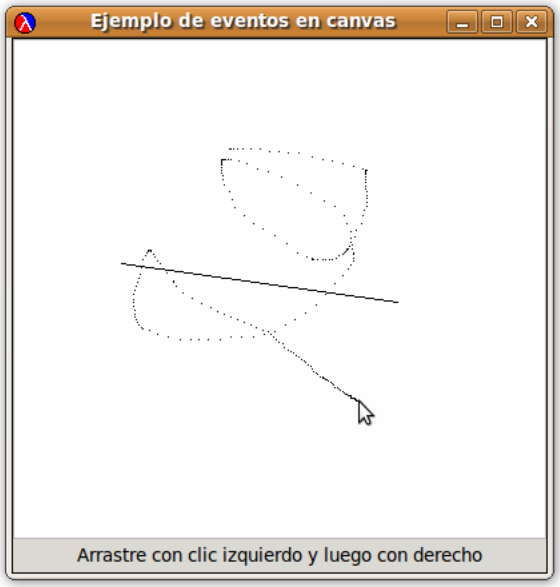


Figura 21.2: canvas2.rkt

```
1  #lang racket
2  ;canvas2.rkt
3  (require racket/gui)
4
```

```
5
6  (define ventana (new frame%
7                       [label "Ejemplo de eventos en canvas"]
8                       [width 400]
9                       [height 400]))
10
11 (define canvas-hijo2%
12   (class canvas%
13
14     (super-new)
15     (define primer-punto (make-object point% 0 0))
16
17     (define/override (on-event evento)
18       (when (send evento button-down? 'left)
19         (send primer-punto set-x (send evento get-x))
20         (send primer-punto set-y (send evento get-y))
21         )
22       (when (send evento button-up? 'left)
23         (send (send this get-dc) draw-line
24               (send primer-punto get-x) (send primer-punto get-y)
25               (send evento get-x) (send evento get-y)
26               )
27         )
28       (when (and (send evento dragging?)
29                  (send evento get-right-down))
30         (send (send this get-dc) draw-point (send evento get-x) (send
               evento get-y))
31         )
32       )
33     ))
34
35 (define c2 (new canvas-hijo2%
36                 [parent ventana]
37                 [style '(border)]
38                 ))
39 (define mensaje (new message%
40                      [parent ventana]
41                      [label "Arrastre con clic izquierdo y luego con
                            derecho"]
42                      ))
43
44 (send ventana show #t)
```

A continuación se implementa el mismo código, pero con memoria:

```
1  #lang racket
2  ;canvas3.rkt
3  (require racket/gui)
4
5
6  (define ventana (new frame%
7                       [label "Ejemplo de canvas con memoria"]
```

Figura 21.3: canvas3.rkt

```
8                       [width 400]
9                       [height 400]))
10
11  ;; Técnica de doble buffer:
12  (define bitmap-de-buffer (make-object bitmap% 400 400))
13  (define dc (make-object bitmap-dc% bitmap-de-buffer))
14
15  (send dc clear) ;Esto inicializa el bitmap
16
17  (define canvas-hijo2%
18    (class canvas%
19
20      (super-new)
21      (define primer-punto (make-object point% 0 0))
22
23      (define/override (on-event evento)
24        (when (send evento button-down? 'left)
25          (send primer-punto set-x (send evento get-x))
26          (send primer-punto set-y (send evento get-y))
27          )
28        (when (send evento button-up? 'left)
29          (send dc draw-line
30                (send primer-punto get-x) (send primer-punto get-y)
31                (send evento get-x) (send evento get-y)
32                )
33          ;Forzar el redibujado en este momento
34          (send this refresh)
```

```
35          )
36        (when (and (send evento dragging?)
37                   (send evento get-right-down))
38          (send dc draw-point (send evento get-x) (send evento get-y))
39          ;Forzar el redibujado en este momento
40          (send this refresh)
41          )
42        )
43      ))
44
45  (define c2 (new canvas-hijo2%
46                  [parent ventana]
47                  [style '(border)]
48                  [paint-callback
49                   (lambda (c dc-canvas)
50                      ;Dibuja el bitmap en el dc-canvas:
51                     (send dc-canvas draw-bitmap bitmap-de-buffer 0 0)
52                     )]
53                  ))
54  (define mensaje (new message%
55                       [parent ventana]
56                       [label "Arrastre con clic izquierdo y luego con
                              derecho"]
57                       ))
58
59  (send ventana show #t)
```

Ahora un ejemplo similar, pero con refresco automático y con memoria:

```
1  #lang racket
2  ;canvas4.rkt
3  (require racket/gui)
4
5  ;; Técnica de doble buffer:
6  (define bitmap-de-buffer (make-object bitmap% 400 400))
7  (define dc (make-object bitmap-dc% bitmap-de-buffer))
8
9  (send dc clear) ;Esto inicializa el bitmap
10
11 ; Lápices:
12 (define negro (make-object color% 0 0 0))
13 (define lápiz-negro-sólido    (make-object pen% negro 2 'solid))
14 (define lápiz-negro-invertido (make-object pen% negro 2 'xor))
15
16 (send dc set-pen lápiz-negro-sólido)
17
18 (define canvas-hijo2%
19   (class canvas%
20
21     (super-new)
22     (define punto-ini-recta (make-object point% 0 0))
23     (define punto-ant-recta (make-object point% 0 0))
```

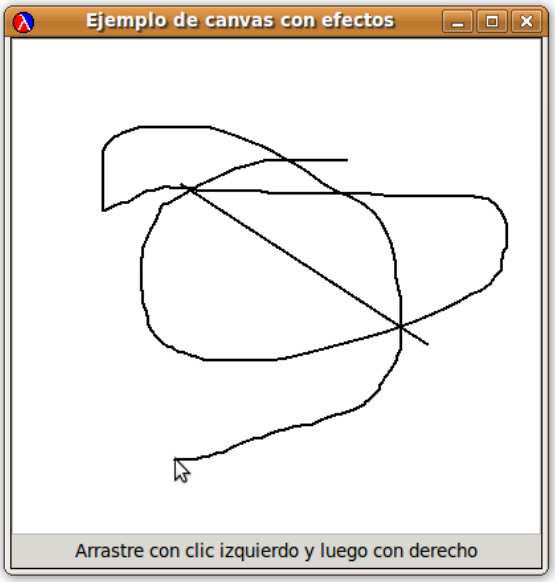


Figura 21.4: canvas4.rkt

```
24      (define punto-ant-lápiz (make-object point% 0 0))
25
26      (define/override (on-event evento)
27        ;BOTÓN DERECHO
28        (when (send evento button-down? 'right)
29          (send punto-ant-lápiz set-x (send evento get-x))
30          (send punto-ant-lápiz set-y (send evento get-y))
31          )
32        (when (and (send evento dragging?)
33                   (send evento get-right-down))
34          (send dc draw-line
35                (send punto-ant-lápiz get-x) (send punto-ant-lápiz get-y)
36                (send evento get-x) (send evento get-y)
37                )
38          (send punto-ant-lápiz set-x (send evento get-x))
39          (send punto-ant-lápiz set-y (send evento get-y))
40          (send this refresh)
41          )
42        ;BOTÓN IZQUIERDO
43        (when (send evento button-down? 'left)
44          (send punto-ini-recta set-x (send evento get-x))
45          (send punto-ini-recta set-y (send evento get-y))
46          (send punto-ant-recta set-x (send evento get-x))
47          (send punto-ant-recta set-y (send evento get-y))
48          )
49        (when (and (send evento dragging?)
```

```
50                  (send evento get-left-down))
51          (send dc set-pen lápiz-negro-invertido)
52          (send dc draw-line
53                  (send punto-ini-recta get-x) (send punto-ini-recta get-y)
54                  (send punto-ant-recta get-x) (send punto-ant-recta get-y)
55                  )
56          (send punto-ant-recta set-x (send evento get-x))
57          (send punto-ant-recta set-y (send evento get-y))
58          (send dc draw-line
59                (send punto-ini-recta get-x) (send punto-ini-recta get-y)
60                (send punto-ant-recta get-x) (send punto-ant-recta get-y)
61                )
62          (send dc set-pen lápiz-negro-sólido)
63          (send this refresh)
64          )
65        (when (send evento button-up? 'left)
66          (send dc draw-line
67                (send punto-ini-recta get-x) (send punto-ini-recta get-y)
68                (send evento get-x) (send evento get-y)
69                )
70          ;Forzar el redibujado en este momento
71          (send this refresh)
72          )
73        )
74      ))
75
76  (define ventana (new frame%
77                       [label "Ejemplo de canvas con efectos"]
78                       [width 400]
79                       [height 400]))
80
81  (define c2 (new canvas-hijo2%
82                  [parent ventana]
83                  [style '(border)]
84                  [paint-callback
85                   (lambda (c dc-canvas)
86                     ;Dibuja el bitmap en el dc-canvas:
87                     (send dc-canvas draw-bitmap bitmap-de-buffer 0 0)
88                     )]
89                  ))
90  (define mensaje (new message%
91                       [parent ventana]
92                       [label "Arrastre con clic izquierdo y luego con
                              derecho"]
93                       ))
94
95  (send ventana show #t)
```

22 Menús

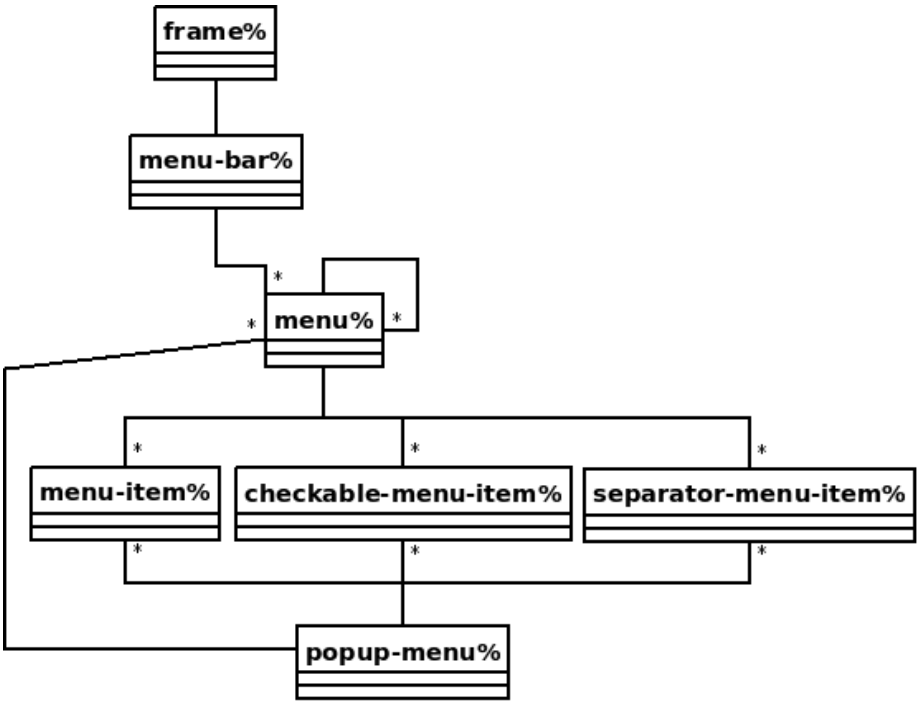


Figura 22.1: Diagrama de clases de los menús en Racket

En la figura 22.1 se muestra el diagrama de clases de las clases relacionadas con menús en Racket.

En la figura 22.2 se ilustra el diagrama de objetos de los menús implementados en el archivo `1-menús.rkt`.

```
1 #lang racket
2 ;1-menús.rkt
3
4 (require racket/gui)
5 (define ventana (new frame% [label "Ejemplo de Menús"]))
6 (define barra-menu (new menu-bar% [parent ventana]
7                          ;[demand-callback (lambda (bm) (printf "evento\n"))
                               ]
```

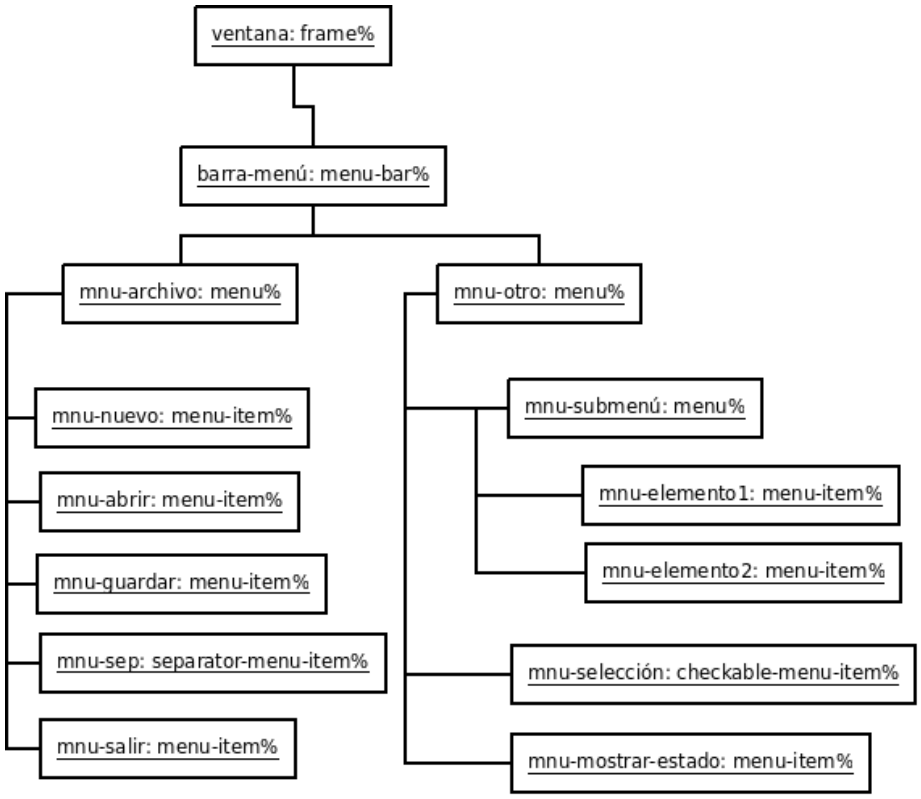


Figura 22.2: Diagrama de objetos del ejemplo 1-menús.rkt

```
8                          ))
9  (define txt-texto (new text-field%
10                         [label #f]
11                         [parent ventana]
12                         [style '(multiple)]
13                         ))
14 (send txt-texto min-width 600)
15 (send txt-texto min-height 600)
16
17 (define mnu-archivo (new menu%
18                           [label "&Archivo"]
19                           [parent barra-menu]))
20 (define mnu-nuevo (new menu-item%
21                         [parent mnu-archivo]
22                         [label "&Nuevo"]
23                         [shortcut #\n]
24                         [callback (lambda (m c)
25                                     (send txt-texto set-value "")
26                                     )]
```

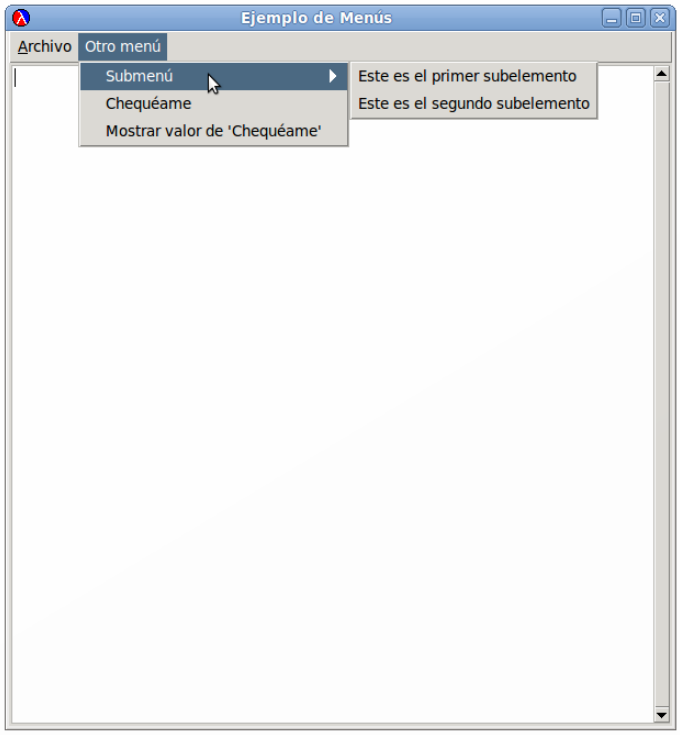


Figura 22.3: 1-menús.rkt

```
27                          ))
28  (define no-hacer-nada (lambda (m c) (void)))
29  (define mnu-abrir (new menu-item%
30                          [parent mnu-archivo]
31                          [label "&Abrir..."]
32                          [shortcut #\a]
33                          [callback no-hacer-nada]
34                          ))
35
36  (define mnu-guardar (new menu-item%
37                           [parent mnu-archivo]
38                           [label "&Guardar..."]
39                           [shortcut #\g]
40                           [callback no-hacer-nada]
41                           ))
42
43  (define mnu-sep (new separator-menu-item% [parent mnu-archivo]))
44  (define mnu-salir (new menu-item%
45                          [parent mnu-archivo]
46                          [label "&Salir"]
47                          [shortcut #\s]
48                          [callback (lambda (m c)
```

```
                                    (send ventana show #f)
                                    )]
                        ))

(define mnu-otro (new menu% [label "Otro menú"][parent barra-menu]))
(define mnu-submenú (new menu% [label "Submenú"][parent mnu-otro]))
(define mnu-elemento1 (new menu-item%
                           [parent mnu-submenú]
                           [label "Este es el primer subelemento"]
                           [callback no-hacer-nada]
                           ))
(define mnu-elemento2 (new menu-item%
                           [parent mnu-submenú]
                           [label "Este es el segundo subelemento"]
                           [callback no-hacer-nada]
                           ))
(define mnu-selección (new checkable-menu-item%
                           [label "Chequéame"]
                           [parent mnu-otro]
                           [callback no-hacer-nada]
                           ))
(define mnu-mostrar-estado
  (new menu-item%
       [label "Mostrar valor de 'Chequéame'"]
       [parent mnu-otro]
       [callback (lambda (m c)
                   (message-box "Ejemplo de 'message-box'"
                                (format "El valor del menú 'Chequéame' es ~
                                    a"
                                        (send mnu-selección is-checked?)))
                   )]
       ))

(send ventana show #t)
```

22.1. Ejemplo de editor sencillo de texto

```
#lang racket
;3-auxiliar.rkt

(require racket/gui)
(provide abrir-archivo guardar-archivo)

(define (abrir-archivo msg par)
  (define ruta (get-file msg par #f #f "txt" null '( ("Archivos de texto"
      "*.txt")("Todos" "*.*") )))
  (if ruta
      (path->string ruta)
      #f
```

```
12          )
13      )
14
15  (define (guardar-archivo msg par)
16      (define ruta (put-file msg par #f #f "txt" null '( ("Archivos de texto"
            "*.txt")("Todos" "*.*") )))
17      (if ruta
18          (path->string ruta)
19          #f
20          )
21      )
```

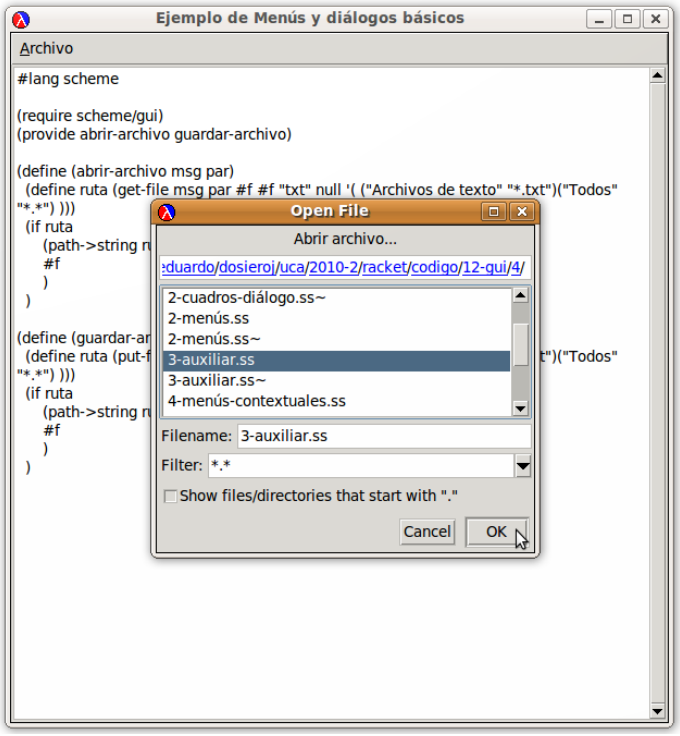


Figura 22.4: 2-menús.rkt

```
1   #lang racket
2   ;2-menús.rkt
3
4   (require racket/gui)
5   (require "3-auxiliar.rkt")
6   (define ventana (new frame% [label "Ejemplo de Menús y diálogos básicos"]))
7   (define barra-menu (new menu-bar% [parent ventana]
8                           ))
9   (define txt-texto (new text-field%
10                         [label #f]
```

```
11                        [parent ventana]
12                        [style '(multiple)]
13                        ))
14  (send txt-texto min-width 600)
15  (send txt-texto min-height 600)
16
17  (define mnu-archivo (new menu%
18                           [label "&Archivo"]
19                           [parent barra-menu]))
20  (define mnu-nuevo (new menu-item%
21                        [parent mnu-archivo]
22                        [label "&Nuevo"]
23                        [shortcut #\n]
24                        [callback (lambda (m c)
25                                    (send txt-texto set-value "")
26                                    )]
27                        ))
28  (define no-hacer-nada (lambda (m c) (void)))
29  (define mnu-abrir (new menu-item%
30                         [parent mnu-archivo]
31                         [label "&Abrir..."]
32                         [shortcut #\a]
33                         [callback
34                          (lambda (m c)
35                            (let ([nombre-archivo (abrir-archivo "Abrir
                                 archivo..." ventana)])
36                              (when nombre-archivo
37                                  (call-with-input-file nombre-archivo
38                                    (lambda (f)
39                                      (define (aux f)
40                                        (let ([cadena (read-string 1000 f)])
41                                          (unless (eof-object? cadena)
42                                            (send txt-texto set-value
43                                                  (string-append (send txt-
                                                      texto get-value)
44                                                                 cadena)
45                                                  )
46                                            )
47                                          )
48                                        )
49                                      (send txt-texto set-value "")
50                                      (aux f)
51                                      )
52                                    )
53                                  )
54                              )
55                            )]
56                         ))
57
58  (define mnu-guardar (new menu-item%
59                           [parent mnu-archivo]
```

```
60                            [label "&Guardar..."]
61                            [shortcut #\g]
62                            [callback
63                             (lambda (m c)
64                               (let ([nombre-archivo (guardar-archivo "Guardar
                                      archivo..." ventana)])
65                                 (if nombre-archivo
66                                     (call-with-output-file nombre-archivo
67                                       (lambda (f)
68                                         (display (send txt-texto get-value) f
                                            )
69                                         )
70                                       )
71                                     (message-box "Error" "No se seleccionó
                                        ningún archivo" ventana)
72                                     )
73                                 )
74                               )]
75                            ))
76
77  (define mnu-sep (new separator-menu-item% [parent mnu-archivo]))
78  (define mnu-salir (new menu-item%
79                         [parent mnu-archivo]
80                         [label "&Salir"]
81                         [shortcut #\s]
82                         [callback (lambda (m c)
83                                     (send ventana show #f)
84                                     )]
85                         ))
86
87
88  (send ventana show #t)
```

22.2. Menús contextuales

```
1   #lang racket
2   ;4-menús-contextuales.rkt
3
4   (require racket/gui)
5   (define ventana (new frame% [label "Ejemplo de Menús Contextuales"]))
6
7   (define brocha (make-object brush% (make-object color% 0 0 255) 'solid))
8   (define mi-canvas%
9     (class canvas%
10      (super-new)
11      (define/override (on-event evento)
12        ;BOTÚN DERECHO
13        (when (send evento button-down? 'right)
14          (send this popup-menu menú (send evento get-x) (send evento get-y))
```

```
15          )
16        )
17
18      ))
19  (define canvas (new mi-canvas%
20                      [parent ventana]
21                      [paint-callback
22                       (lambda (c dc)
23                         (define-values (ancho alto) (send dc get-size))
24                         (send dc set-brush brocha)
25                         (send dc draw-rectangle 0 0 ancho alto)
26                         )]
27                      ))
28
29  (send ventana min-width 600)
30  (send ventana min-height 600)
31
32  (define (imprime-menú m c)
33    (printf "Opción seleccionada: '~a'\n" (send m get-label))
34    )
35
36  (define menú (new popup-menu% ))
37  (define opción1 (new menu-item%
38                       [parent menú]
39                       [label "Opción 1"]
40                       [callback imprime-menú]
41                       ))
42  (define opción2 (new menu-item%
43                       [parent menú]
44                       [label "Opción 2"]
45                       [callback imprime-menú]
46                       ))
47  (define opción3 (new menu-item%
48                       [parent menú]
49                       [label "Opción 3"]
50                       [callback imprime-menú]
51                       ))
52
53
54  (send ventana show #t)
```

Otro ejemplo con menús contextuales:

```
1  #lang racket
2  ;5-selección-color.rkt
3
4  (require racket/gui)
5
6  ;; Técnica de doble buffer:
7  (define bitmap-de-buffer (make-object bitmap% 400 400))
8  (define dc (make-object bitmap-dc% bitmap-de-buffer))
9
```

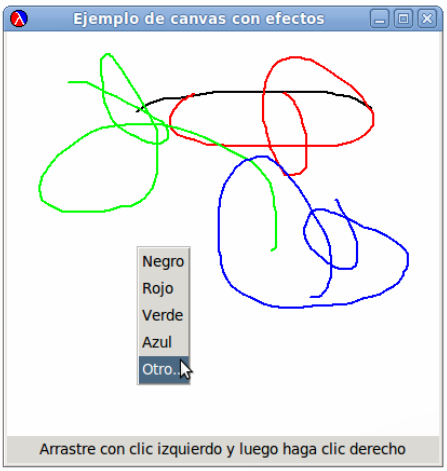


Figura 22.5: 5-selección-color.rkt - menú

```
10 (send dc clear) ;Esto inicializa el bitmap
11
12 ; Lápices:
13 (define negro (make-object color% 0 0 0))
14 (define lápiz-sólido    (make-object pen% negro 2 'solid))
15 (define lápiz-invertido (make-object pen% negro 2 'xor))
16
17 (send dc set-pen lápiz-sólido)
18
19 (define canvas-hijo2%
20   (class canvas%
21
22     (super-new)
23     (define punto-ant-lápiz (make-object point% 0 0))
24
25     (define/override (on-event evento)
26       ;BOTÓN DERECHO
27       (when (send evento button-down? 'right)
28         (send this popup-menu menú (send evento get-x) (send evento get-y))
29         )
30       ;BOTÓN IZQUIERDO
31       (when (send evento button-down? 'left)
32         (send punto-ant-lápiz set-x (send evento get-x))
33         (send punto-ant-lápiz set-y (send evento get-y))
34         )
35       (when (and (send evento dragging?)
36                  (send evento get-left-down))
37         (send dc draw-line
38               (send punto-ant-lápiz get-x) (send punto-ant-lápiz get-y)
39               (send evento get-x) (send evento get-y)
```

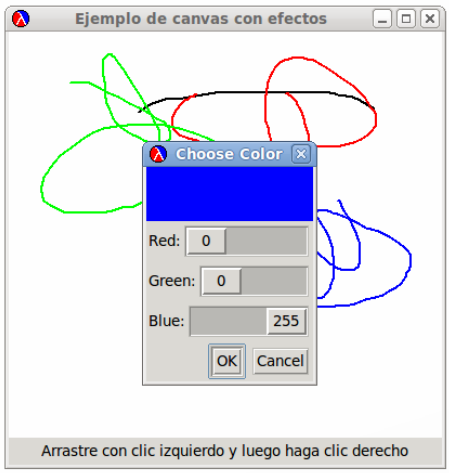


Figura 22.6: 5-selección-color.rkt - Selector de color 1

```
40                )
41          (send punto-ant-lápiz set-x (send evento get-x))
42          (send punto-ant-lápiz set-y (send evento get-y))
43          (send this refresh)
44          )
45        )
46      ))
47
48  (define ventana (new frame%
49                       [label "Ejemplo de canvas con efectos"]
50                       [width 400]
51                       [height 400]))
52
53  (define c2 (new canvas-hijo2%
54                  [parent ventana]
55                  ;[style '(border)]
56                  [paint-callback
57                   (lambda (c dc-canvas)
58                     ;Dibuja el bitmap en el dc-canvas:
59                     (send dc-canvas draw-bitmap bitmap-de-buffer 0 0)
60                     )]
61                  ))
62  (define mensaje (new message%
63                       [parent ventana]
64                       [label "Arrastre con clic izquierdo y luego haga clic
                              derecho"]
65                       ))
66
67
68  (define menú (new popup-menu% ))
```

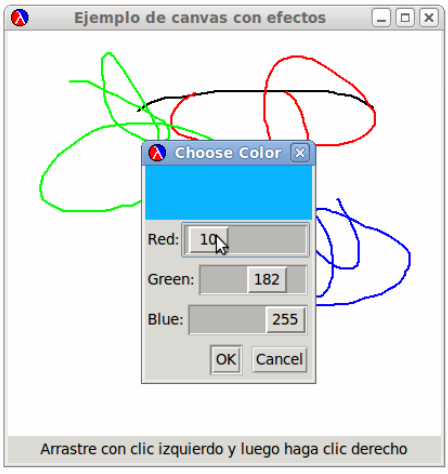


Figura 22.7: 5-selección-color.rkt- Selector de color 2

```
69 (define opción0 (new menu-item%
70                      [parent menú]
71                      [label "Negro"]
72                      [callback
73                       (lambda (m c)
74                         (send dc set-pen lápiz-invertido)
75                         (send lápiz-sólido set-color (make-object color% 0
                               0 0))
76                         (send dc set-pen lápiz-sólido)
77                         )]
78                      ))
79 (define opción1 (new menu-item%
80                      [parent menú]
81                      [label "Rojo"]
82                      [callback
83                       (lambda (m c)
84                         (send dc set-pen lápiz-invertido)
85                         (send lápiz-sólido set-color (make-object color%
                               255 0 0))
86                         (send dc set-pen lápiz-sólido)
87                         )]
88                      ))
89 (define opción2 (new menu-item%
90                      [parent menú]
91                      [label "Verde"]
92                      [callback
93                       (lambda (m c)
94                         (send dc set-pen lápiz-invertido)
95                         (send lápiz-sólido set-color (make-object color% 0
                               255 0))
```

```
96                          (send dc set-pen lápiz-sólido)
97                          )]
98                        ))
99  (define opción3 (new menu-item%
100                       [parent menú]
101                       [label "Azul"]
102                       [callback
103                        (lambda (m c)
104                          (send dc set-pen lápiz-invertido)
105                          (send lápiz-sólido set-color (make-object color% 0
                                 0 255))
106                          (send dc set-pen lápiz-sólido)
107                          )]
108                       ))
109 (define opción4 (new menu-item%
110                       [parent menú]
111                       [label "Otro..."]
112                       [callback
113                        (lambda (m c)
114                          (send dc set-pen lápiz-invertido)
115                          ;(send lápiz-sólido set-color (make-object color% 0
                               0 255))
116                          (let ([nuevo-color (get-color-from-user "Elija un
                              color" ventana (send lápiz-sólido get-color))])
117                            (when nuevo-color
118                                (send lápiz-sólido set-color nuevo-color)
119                                )
120                            )
121                          (send dc set-pen lápiz-sólido)
122                          )]
123                       ))
124
125 (send ventana show #t)
```

23 Proyecto: Minipaint

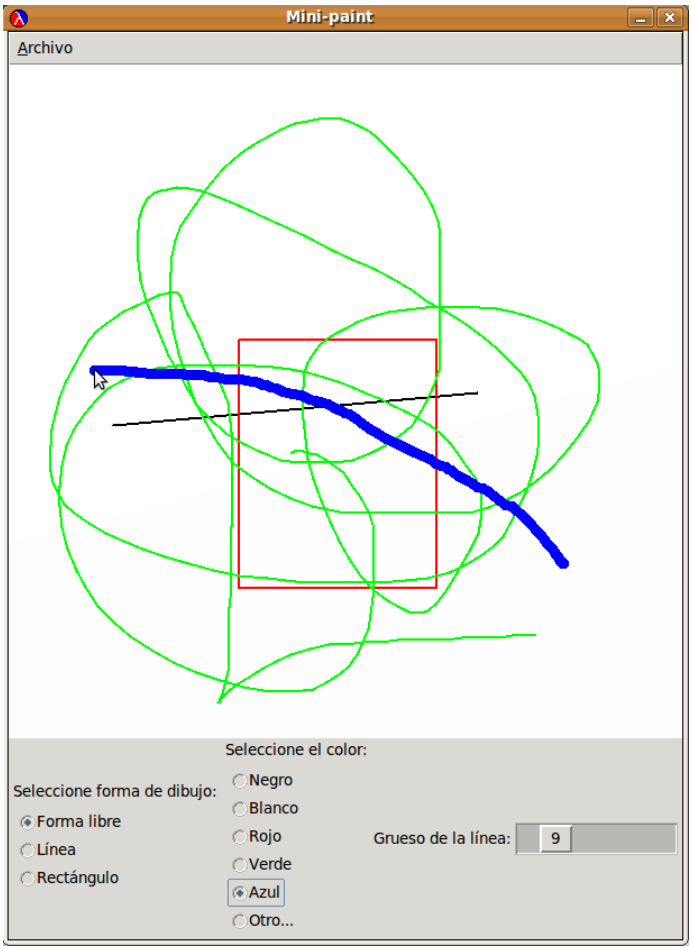


Figura 23.1: mini-paint.rkt

```
1  #lang racket
2  ;mini-paint.rkt
3
4  (require racket/gui)
5
6  (define ANCHO 600)
```

```
7  (define ALTO 600)
8
9  (define lista-de-formas (list 'libre 'línea 'rectángulo))
10
11 (define forma (list-ref lista-de-formas 0))
12
13 (define nombre-del-archivo #f)
14 (define modificado? #f)
15
16 ;; Técnica de doble buffer:
17 (define bitmap-de-buffer (make-object bitmap% ANCHO ALTO))
18 (define dc (make-object bitmap-dc% bitmap-de-buffer))
19 (send dc clear) ;Inicializar el bitmap
20
21 ; Lápices y brocha:
22 (define color-sólido (make-object color% 0 0 0))
23 (define lápiz-sólido    (make-object pen% color-sólido 2 'solid))
24 (define lápiz-invertido (make-object pen% color-sólido 2 'xor))
25 (define brocha (make-object brush% color-sólido 'transparent))
26
27 (send dc set-pen lápiz-sólido)
28 (send dc set-brush brocha)
29
30 (define (menor-y-diferencia-absoluta a b)
31   (values (min a b) (abs (- a b)))
32   )
33
34 (define canvas-hijo%
35   (class canvas%
36
37     (super-new)
38     (define punto-inicial (make-object point% 0 0));Punto sobre el que hizo
           clic al principio.
39     (define punto-anterior (make-object point% 0 0));Punto anterior donde
          estuvo el ratón.
40     (define rect-x 0)
41     (define rect-y 0)
42     (define rect-ancho 0)
43     (define rect-alto 0)
44
45     (define (actualizar-rectángulo)
46       #| Actualiza los valores de rect-x, rect-y, rect-ancho y rect-alto
47 en función de los puntos punto-inicial y punto-anterior.|#
48       (set!-values
49        (rect-x rect-ancho)
50        (menor-y-diferencia-absoluta
51         (send punto-inicial get-x) (send punto-anterior get-x)
52         )
53        )
54       (set!-values
55        (rect-y rect-alto)
```

```
56          (menor-y-diferencia-absoluta
57           (send punto-inicial get-y) (send punto-anterior get-y)
58           )
59          )
60         )
61
62       (define/override (on-event evento)
63         ;;Forma libre:
64         (when (equal? forma 'libre)
65           (when (send evento button-down? 'left)
66             (send punto-anterior set-x (send evento get-x))
67             (send punto-anterior set-y (send evento get-y))
68             (set! modificado? #t)
69             )
70           (when (and (send evento dragging?)
71                      (send evento get-left-down))
72             (send dc draw-line
73                   (send punto-anterior get-x) (send punto-anterior get-y)
74                   (send evento get-x) (send evento get-y)
75                   )
76             (send punto-anterior set-x (send evento get-x))
77             (send punto-anterior set-y (send evento get-y))
78             (send this refresh)
79             )
80           )
81         ;;Línea:
82         (when (equal? forma 'línea)
83           (when (send evento button-down? 'left)
84             (send punto-inicial set-x (send evento get-x))
85             (send punto-inicial set-y (send evento get-y))
86             (send punto-anterior set-x (send evento get-x))
87             (send punto-anterior set-y (send evento get-y))
88             (send dc set-pen lápiz-invertido)
89             (set! modificado? #t)
90             )
91           (when (and (send evento dragging?)
92                      (send evento get-left-down))
93             (send dc draw-line
94                   (send punto-inicial get-x) (send punto-inicial get-y)
95                   (send punto-anterior get-x) (send punto-anterior get-y)
96                   )
97             (send punto-anterior set-x (send evento get-x))
98             (send punto-anterior set-y (send evento get-y))
99             (send dc draw-line
100                  (send punto-inicial get-x) (send punto-inicial get-y)
101                  (send punto-anterior get-x) (send punto-anterior get-y)
102                  )
103            (send this refresh)
104            )
105          (when (send evento button-up? 'left)
106            (send dc set-pen lápiz-sólido)
```

```
107             (send dc draw-line
108                   (send punto-inicial get-x) (send punto-inicial get-y)
109                   (send evento get-x) (send evento get-y)
110                   )
111             ;Forzar el redibujado en este momento
112             (send this refresh)
113             )
114           )
115         ;;Rectángulo:
116         (when (equal? forma 'rectángulo)
117           (when (send evento button-down? 'left)
118             (send punto-inicial set-x (send evento get-x))
119             (send punto-inicial set-y (send evento get-y))
120             (send punto-anterior set-x (send evento get-x))
121             (send punto-anterior set-y (send evento get-y))
122             (set! rect-x (send evento get-x))
123             (set! rect-y (send evento get-x))
124             (set! rect-ancho 0)
125             (set! rect-alto 0)
126             (send dc set-pen lápiz-invertido)
127             (set! modificado? #t)
128             )
129           (when (and (send evento dragging?)
130                      (send evento get-left-down))
131             (send dc draw-rectangle rect-x rect-y rect-ancho rect-alto)
132
133             (send punto-anterior set-x (send evento get-x))
134             (send punto-anterior set-y (send evento get-y))
135             (actualizar-rectángulo)
136
137             (send dc draw-rectangle
138                   rect-x rect-y rect-ancho rect-alto)
139             (send this refresh)
140             )
141           (when (send evento button-up? 'left)
142             (send dc set-pen lápiz-sólido)
143             (send punto-anterior set-x (send evento get-x))
144             (send punto-anterior set-y (send evento get-y))
145             (actualizar-rectángulo)
146
147             (send dc draw-rectangle
148                   rect-x rect-y rect-ancho rect-alto)
149
150             ;Forzar el redibujado en este momento
151             (send this refresh)
152             )
153           )
154         )
155       ))
156
157   (define frame-hijo%
```

```
158   (class frame%
159     (super-new)
160     (define/augment (on-close)
161       (send mnu-salir command (new control-event% [event-type 'menu]))
162       )
163     ))
164 
165 (define ventana (new frame-hijo%
166                      [label "Mini-paint"]
167                      [stretchable-width #f]
168                      [stretchable-height #f]
169                      ))
170 
171 (define mi-canvas (new canvas-hijo%
172                         [parent ventana]
173                         [min-width ANCHO]
174                         [min-height ALTO]
175                         [style '(no-focus)]
176                         [paint-callback
177                          (lambda (c dc-canvas)
178                            ;Dibuja el bitmap en el dc-canvas:
179                            (send dc-canvas draw-bitmap bitmap-de-buffer 0 0)
180                            )]
181                        ))
182 (define panel (new horizontal-panel% [parent ventana]))
183 (define radio-forma (new radio-box%
184                          [label "Seleccione forma de dibujo:"]
185                          [parent panel]
186                          [choices (list "Forma libre" "Línea" "Rectángulo")
                                 ]
187                          [callback
188                           (lambda (r c)
189                             (set! forma (list-ref lista-de-formas (send r
                                     get-selection)))
190                             )]
191                          [style (list 'vertical 'vertical-label)]
192                          ))
193 (define radio-color (new radio-box%
194                          [label "Seleccione el color:"]
195                          [parent panel]
196                          [choices (list "Negro" "Blanco" "Rojo" "Verde" "
                                 Azul" "Otro...")]
197                          [callback
198                           (lambda (r c)
199                             (send dc set-pen lápiz-invertido)
200                             (send lápiz-sólido set-color
201                                   (case (send r get-selection)
202                                     [(0) (make-object color% 0 0 0)]
203                                     [(1) (make-object color% 255 255 255)]
204                                     [(2) (make-object color% 255 0 0)]
205                                     [(3) (make-object color% 0 255 0)]
```

```
206                                     [(4) (make-object color% 0 0 255)]
207                                     [(5)
208                                      (let ([nuevo-color
209                                             (get-color-from-user "Elija un
                                                 color" ventana (send lápiz-
                                                 sólido get-color))])
210                                        (if nuevo-color nuevo-color (send
                                               lápiz-sólido get-color))
211                                        )
212                                      ]
213                                     )
214                                   )
215                                (send dc set-pen lápiz-sólido)
216                                (send lápiz-invertido set-color (send lápiz-
                                       sólido get-color))
217                                )]
218                             [style (list 'vertical 'vertical-label)]
219                             ))
220
221  (define barra-grueso (new slider%
222                        [label "Grueso de la línea:"]
223                        [parent panel]
224                        [min-value 1]
225                        [max-value 50]
226                        [init-value 2]
227                        [callback
228                         (lambda (s e)
229                           (send lápiz-invertido set-width (send s get-value))
230                           (send dc set-pen lápiz-invertido)
231                           (send lápiz-sólido set-width (send s get-value))
232                           (send dc set-pen lápiz-sólido)
233                           )]
234                        ))
235
236  (define filtro '( ("Archivos de imagen PNG" "*.png")("Todos" "*.*")))
237  (define (abrir-archivo msg par)
238    (define ruta (get-file msg par #f #f "png" null filtro ))
239    (if ruta
240        (path->string ruta)
241        #f
242        )
243    )
244
245  (define (guardar-archivo msg par)
246    (define ruta (put-file msg par #f #f "png" null filtro))
247    (if ruta
248        (path->string ruta)
249        #f
250        )
251    )
252
```

```
(define barra-menu (new menu-bar% [parent ventana]))
(define mnu-archivo (new menu%
                         [label "&Archivo"]
                         [parent barra-menu]))
(define mnu-nuevo (new menu-item%
                        [parent mnu-archivo]
                        [label "&Nuevo"]
                        [shortcut #\n]
                        [callback (lambda (m c)
                                    (send dc clear)
                                    (send mi-canvas refresh)
                                    (set! nombre-del-archivo #f)
                                    )]
                        ))
(define mnu-abrir (new menu-item%
                        [parent mnu-archivo]
                        [label "&Abrir..."]
                        [shortcut #\a]
                        [callback
                         (lambda (m c)
                           (when (or (not modificado?)
                                     (equal? 1
                                             (message-box/custom
                                              "Advertencia"
                                              "Si abre un nuevo archivo
                                                  perderá los cambios
                                                  realizados"
                                              "Abrir archivo"
                                              "Cancelar"
                                              #f
                                              ventana
                                              '(caution disallow-close
                                                  default=2)
                                              )))
                             (let ([nombre-archivo (abrir-archivo "Abrir
                                 imagen..." ventana)])
                               (when nombre-archivo
                                 (let ([nuevo-buffer (make-object bitmap%
                                     1 1)])
                                   (send dc set-pen "black" 1 'solid);
                                       desligando el lápiz-sólido
                                   (send dc set-brush "black" 'solid);
                                       desligando la brocha
                                   (send nuevo-buffer load-file nombre-
                                       archivo );abrir archivo
                                   (set! dc (make-object bitmap-dc% nuevo-
                                       buffer));nuevo dc
                                   (set! bitmap-de-buffer nuevo-buffer);
                                       usar el nuevo
                                   (send dc set-pen lápiz-sólido);
                                       configurar el lápiz
```

```
293                                     (send dc set-brush brocha);configurar
                                            la brocha
294                                     )
295                                   (set! nombre-del-archivo nombre-archivo)
296                                   (set! modificado? #f)
297                                   (send mi-canvas refresh)
298                                   )
299                                 )
300                               )
301                             )]
302                         ))
303
304 (define mnu-guardar (new menu-item%
305                          [parent mnu-archivo]
306                          [label "&Guardar..."]
307                          [shortcut #\g]
308                          [callback
309                           (lambda (m c)
310                             (when modificado?
311                               (if nombre-del-archivo
312                                   (begin
313                                     (send bitmap-de-buffer save-file nombre
                                            -del-archivo 'png)
314                                     (set! modificado? #f)
315                                     )
316                                   (let ([nombre-archivo (guardar-archivo "
                                        Guardar archivo..." ventana)])
317                                     (if nombre-archivo
318                                         (begin
319                                           (send bitmap-de-buffer save-file
                                                nombre-archivo 'png)
320                                           (set! modificado? #f)
321                                           (set! nombre-del-archivo nombre-
                                                archivo)
322                                           )
323                                         (message-box "Error" "No se
                                            seleccionó ningún nombre"
                                            ventana)
324                                         )
325                                     )
326                                   )
327                               )
328                             )]
329                          ))
330
331 (define mnu-sep (new separator-menu-item% [parent mnu-archivo]))
332 (define mnu-salir (new menu-item%
333                        [parent mnu-archivo]
334                        [label "&Salir"]
335                        [shortcut #\s]
336                        [callback
```

```
337                          (lambda (m c)
338                            (when (and modificado?
339                                       (equal? 1
340                                               (message-box/custom
341                                                "Advertencia"
342                                                "No ha guardado el archivo
                                                       actual"
343                                                "Guardar archivo"
344                                                "No guardar"
345                                                #f
346                                                ventana
347                                                '(caution disallow-close
                                                       default=1)
348                                                )))
349                              (if nombre-del-archivo
350                                  (send bitmap-de-buffer save-file nombre-del
                                         -archivo 'png)
351                                  (let ([nombre-archivo (guardar-archivo "
                                         Guardar archivo..." ventana)])
352                                    (if nombre-archivo
353                                        (send bitmap-de-buffer save-file
                                               nombre-archivo 'png)
354                                        (void)
355                                        )
356                                    )
357                                  )
358                              )
359                            (send ventana show #f)
360                            )]
361                          ))
362
363  (send ventana show #t)
```

Ejercicios de Interfaces Gráficas de Usuario

1. Elaborar un programa que muestre una ventana que contenga una etiqueta, pidiendo el nombre de una persona y una caja de texto donde se escribirá el nombre, y un botón "enviar" cuando se presione el botón, se creará una segunda ventana mostrando un saludo unido con el nombre que fue escrito en la primer ventana y un botón "regresar". Cuando la segunda ventana se cree ella tendrá el foco y la primer ventana no se podrá acceder, a menos que se presione el botón "regresar" de la segunda ventana, que cerrará la ventana dos y de nuevo estará habilitada la primer ventana.
2. Elaborar un programa que muestre una ventana que tenga como nombre "Encriptamiento de cadenas", esta ventana tendrá una caja de texto que diga que debe de ingresar una cadena, una segunda caja de texto que diga cadena encriptada, y tres botones, el primer botón "Encriptar" que al ser presionado encriptará la cadena ingresada en la primer caja de texto y la mostrará en la segunda caja de texto, si al presionar el botón de "Encriptar" la primera caja de texto esta vacía, mostrar un ventana que indique que la caja de texto esta vacía. El segundo botón "Limpiar" borrará la información que está en ambas cajas de texto. Y el botón "Salir" que terminará el programa.
3. Elaborar un programa que muestre una ventana que contenga al lado izquierdo una caja de texto con una etiqueta "agregar hijo" y un botón "agregar" y del lado derecho de la ventana una lista que originalmente esté vacía, al escribir una cadena en la caja de texto y al presionar el botón, la cadena que fue escrita en la caja de texto se agregará a la lista del lado derecho de la ventana, quitando la información de la caja de texto, dejándola preparada para un nuevo valor, se debe validar que si la caja de texto está vacía no se podrá agregar algo a la lista, esta validación se hará mostrando una ventana que lo indique.
4. Elaborar un programa que muestre una ventana que contenga como título "Cargar imagen", y un botón que diga "Cargar" este botón abrirá un cuadro de diálogo, para poder buscar archivos de imágenes, seleccionar una imagen y al abrirla, abrir una nueva ventana, donde se muestre la imagen seleccionada, y junto a un botón de regresar, que regresará a la ventana inicial.
5. Elaborar un programa que muestre una ventana que contenga una barra de menús, con el menú Archivo, que contendrá como hijos: nuevo, abrir, guardar y salir. La ventana

inicialmente tendrá un área de texto vacía donde se pueda escribir, al presionar el menú archivo y la opción de nuevo, el contenido del área de texto se borrará, dejando limpia el área de texto, si se presiona la opción abrir, se abrirá un cuadro de dialogo para poder buscar y seleccionar una archivo de texto, luego al seleccionar y abrir el archivo seleccionado, la información se cargará en el área de texto, para poder editarse, si se desea o si sólo se desea ver el contenido. Si se elige la opción de guardar se abrirá un cuadro de dialogo para poder guardar el archivo en un lugar del disco duro, donde se desee. Y al presionar la opción de salir, el programa terminará.

6. Elaborar un programa que muestre una ventana que contenga un list-box de 5 elementos (elementos de su elección) y un botón "seleccionar", cuando se dé clic a dicho botón, deberá aparecer una ventana emergente con el elemento seleccionado al centro de la ventana y un botón salir que permitirá cerrar la ventana emergente.

7. Realizar un programa que muestre una ventana cuyo título será "formulario" en esta se presentará un formulario donde el usuario ingresará los siguientes datos:

   a) - Nombre (text-field)
   b) - edad (combo-box edad máxima 90 años)
   c) - sexo (radio-button M F)
   d) - botón finalizar

   El botón finalizar cerrará la ventana "formulario" y abrirá la ventana "revisado" donde mostrará los datos ingresados por el usuario, esta ventana presentará dos botones "guardar" y "salir".
   "guardar": Los datos serán guardados en un archivo.
   "salir": cerrará el programa en caso de no guardar enviar un mensaje que indique "datos no almacenados".

8. Realizar un programa que muestre la ventana cuyo título será "países" esta presentará un list-box que contendrá una lista de países almacenados en el archivo "países.txt" y presentará dos botones "eliminar" "salir".
   "eliminar": eliminará el país seleccionado de la lista y del archivo.
   "salir" : cerrará la ventana.

9. Realizar un programa que muestre una ventana cuyo título será "Seleccionar año", esta ventana presentará un slider que tendrá una lista de años desde el año 1900 hasta el año 3000 y un botón "seleccionar", al dar clic al botón "seleccionar" aparecerá una ventana cuyo título será el año seleccionado, esta ventana contendrá dos text-field "nombre" y "suceso" en donde se almacenará el nombre de la persona que digita la información y el suceso ocurrido en el año seleccionado finalmente la ventana presentará el botón "guardar" lo cual permitirá almacenar la información en un archivo. Nota la ventana "seleccionar año" no debe cerrarse sino simplemente quedar inactiva mientras se ingresan los datos en la segunda ventana, al momento de guardar la in-

formación, la ventana emergente se cerrará y se podrá utilizar la ventana "seleccionar año" nuevamente.

Parte V

Apéndices

A Diferencias entre PLT Scheme y Racket

Las diferencias radican en lo siguiente:

1. *Racket* es el nuevo nombre de *PLT Scheme*, comenzando con la versión 5.0. Lo que significa que "Racket 5.0" equivale a "PLT Scheme 5.0".
2. Las extensiones tradicionales de PLT Scheme son `.ss` y `.scm`. Con Racket ahora se prefieren las extensiones `.rkt`.
3. El lenguaje principal del intérprete de PLT Scheme se llama `scheme` y no `racket` como en Racket. Por ello, la primera línea de los programas de PLT Scheme deben comenzar con la línea: `#lang scheme`, en lugar de `#lang racket` como en Racket.
4. El ejecutable de DrRacket en versiones anteriores a la 5.0 es `drscheme` (y se llama DrScheme) en lugar de `drracket`.
5. El ejecutable de la herramienta de consola de Racket en versiones anteriores a la 5.0 es `mzscheme` en lugar de `racket`.
6. El intérprete de los programas Racket con Interfaz Gráfica de Usuario en versiones anteriores a la 5.0 es `mred` en lugar de `gracket`.
7. La compilación con PLT Scheme se realiza con el comando `mzc` en lugar de `raco`:
   `$ mzc --exe <nom-ejecutable> <archivo-fuente>.ss`
   para programas sin interfaz gráfica, y con:
   `$ mzc --gui-exe <nom-ejecutable> <archivo-fuente>.ss`
   para programas con interfaz gráfica.
8. En PLT Scheme, el módulo de interfaces gráficas de usuario se llama `scheme/gui`.

Para mayor información sobre el cambio de nombre, refiérase al sitio:
http://racket-lang.org/new-name.html.

-

Bibliografía

[1] http://docs.racket-lang.org/ (2009-2010) - *Sitio de documentación oficial de Racket*, por la Comunidad de desarrolladores de Racket (http://racket-lang.org/people.html).

[2] http://www.htdp.org/2003-09-26/Book/ - *How to Design Programs (An Introduction to Computing and Programming)*, por Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi. The MIT Press, Massachusetts Institute of Technology.