

TEORÍA DE LENGUAJES DE PROGRAMACIÓN



# RACKET PROGRAMACIÓN FUNCIONAL

ANDREA LAZZARETTO    MARÍA PEINADO TOLEDO

YOLANDA SARMIENTO RINCÓN    PEDRO SÁNCHEZ MACHUCA

# ÍNDICE

## 01 Introducción

1.1 Filosofía detrás	03
1.2 Lenguajes funcionales	03
1.3 Historia de Racket	07
1.4 Soporte para DSL	09

## 03 Sistema de Tipos

3.1 Tipos dinámicos	14
3.2 Tipos Estáticos	15
3.3 Comparación	17
3.4 Tipado Gradual	18

## 05 Casos de uso y ejemplos

5.1 Racket en la docencia	27
5.2 Ventajas de su uso	28
5.3 Ejemplos de su uso en la docencia	28

## 02 Sintaxis basada en S-expresiones

2.1 Definición y características	10
2.2 Cómo se utilizan	11

## 04 Módulos, Macros y su uso para la creación de Lenguajes Específicos del Dominio

4.1 Módulos en Racket	19
4.2 Macros en Racket	21
4.3 Racket como Lenguaje orientado a lenguajes	22

## 06 Bibliografía

# INTRODUCCIÓN

## 1.1. Filosofía detrás de Racket

---

Racket es un lenguaje funcional concebido como una plataforma para crear y experimentar con nuevos lenguajes, basándose en la idea de que cada lenguaje debe ser una herramienta adaptada a problemas o dominios específicos. Esto lo distingue de otros lenguajes funcionales, a menudo centrados en la optimización del paradigma funcional en lugar de en la extensibilidad.

Además, Racket nació con un fuerte propósito educativo, facilitando la comprensión de conceptos avanzados como la creación de lenguajes, la manipulación de código y la programación funcional.

## 1.2. Lenguajes funcionales

---

Antes de poder empezar a tratar Racket es fundamental comprender qué son los lenguajes funcionales

Un lenguaje de programación funcional es un paradigma de programación que se basa en funciones matemáticas para resolver problemas. Este paradigma se concentra en el "qué hacer" en lugar de "cómo hacerlo", haciéndolo diferente de los paradigmas imperativos como C o Java, que especifican instrucciones detalladas para manipular el estado del programa.

La diferencia principal está, por lo tanto, en el nivel de abstracción. En los lenguajes funcionales como Racket, trabajas en un nivel más alto: describes qué quieres obtener (por ejemplo, "sumar todos los números") y el lenguaje proporciona herramientas ya preparadas para ejecutar el trabajo. En los lenguajes imperativos como C, en cambio, debes especificar detalladamente cada paso para alcanzar el mismo objetivo.

Consideremos el siguiente ejemplo:

Queremos calcular el producto de los números pares en una lista:

## En Racket:

```
(define lista '(1 2 3 4 5 6))  
  
; Filtra los números pares y luego los multiplica  
(define resultado (foldl * 1 (filter even? lista)))  
  
(displayln resultado) ; Devuelve 48 (2 * 4 * 6)
```

- La función filter selecciona solo los números pares de la lista.
- La función foldl multiplica todos los números filtrados.
- El programador se concentra solo en qué quiere hacer (filtrar y multiplicar).

## En C:

```
#include <stdio.h>  
  
int multiplicar_pares(int numeros[], int tamano) {  
    int resultado = 1; // Variable para acumular el resultado  
    for (int i = 0; i < tamano; i++) {  
        if (numeros[i] % 2 == 0) { // Verifica si el número es par  
            resultado *= numeros[i]; // Multiplica el resultado actual  
        }  
    }  
    return resultado; // Devuelve el resultado  
}  
  
int main() {  
    int numeros[] = {1, 2, 3, 4, 5, 6};  
    printf("Resultado: %d\n", multiplicar_pares(numeros, 6)); // Imprime el resultado  
    return 0;  
}
```

- Debes iterar sobre la lista manualmente con un ciclo.
- Debes comprobar manualmente cada elemento (if (numeros[i] % 2 == 0)).
- Debes actualizar manualmente la variable resultado.

## Características principales de un lenguaje funcional:

### 1.Funciones como ciudadanos de primera clase:

En los lenguajes funcionales, las funciones son consideradas valores de primera clase. Esto significa que puedes:

- Pasarlas como argumentos a otras funciones.
- Devolverlas como resultado desde una función.
- Asignarlas a variables.

```
; Definición de una función que toma otra función como argumento  
(define (aplicar-funcion f x)  
  (f x)) ; Aplica la función f al valor x  
  
; Una función anónima que calcula el cuadrado de un número  
(define cuadrado (lambda (x) (* x x)))  
  
; Uso de la función  
(displayln (aplicar-funcion cuadrado 5)) ; Devuelve 25
```

- aplica-función es una función que acepta otra función f y un valor x, aplicando f a x.
- cuadrado es una función anónima (definida con lambda) que calcula el cuadrado de un número.
- Este ejemplo demuestra cómo las funciones pueden ser pasadas y utilizadas como valores.

## 2. Higher-order functions (funciones de orden superior):

Las funciones de orden superior son funciones que aceptan otras funciones como entrada o las devuelven como salida.

<pre>; Función de orden superior que aplica una función a cada elemento de una lista (define (aplicar-a-todos f lista)   (map f lista)) ; Usa map para aplicar f a cada elemento  ; Una función anónima para calcular el cuadrado (define lista '(1 2 3 4)) (displayln (aplicar-a-todos (lambda (x) (* x x)) lista)) ; Devuelve '(1 4 9 16)</pre>	aplica-a-todos utiliza map para transformar cada elemento de la lista aplicando la función f.
---	---

## 3. Inmutabilidad

En los lenguajes funcionales, los datos son a menudo inmutables, es decir, no pueden ser modificados después de ser creados. En lugar de cambiar los datos existentes, se crean nuevos valores.

<pre>; Lista original (inmutable) (define lista-original '(1 2 3))  ; Creación de una nueva lista agregando un elemento (define nueva-lista (cons 0 lista-original)) ; '(0 1 2 3)  ; Imprime ambas listas (displayln lista-original) ; Devuelve '(1 2 3) (displayln nueva-lista) ; Devuelve '(0 1 2 3)</pre>	<ul style="list-style-type: none"><li>- cons crea una nueva lista añadiendo un elemento (0) al inicio de lista-original.</li><li>- lista-original no se modifica; al contrario, se crea una nueva lista (nueva-lista).</li></ul>
--	--

## 4. Funciones puras

Los lenguajes funcionales enfatizan el uso de funciones puras, no es un requisito absoluto pero se intenta lograrlo. Una función para ser definida como pura debe cumplir los siguientes criterios:

- Determinismo: Dado el mismo input, siempre devuelve el mismo output.
- Ausencia de efectos secundarios: No modifica el estado externo, como variables globales, archivos o la memoria.

Ejemplo de una función pura

```
(define (cuadrado x)
  (* x x)) ; No interactúa con el mundo exterior y siempre devuelve el mismo resultado

(displayln (cuadrado 4)) ; Devuelve 16
```

¿Qué es una función impura?

Una función es impura si:

- Produce efectos secundarios, como modificar variables globales, escribir en archivos o leer la entrada del usuario.
- Su resultado puede depender de un estado externo o de una interacción con el mundo exterior.

<pre>(define contador 0) ; Variable global  (define (incrementar-contador)   (set! contador (+ contador 1)) ; Modifica la variable global   contador) ; Devuelve el nuevo valor de contador  (displayln (incrementar-contador)) ; Devuelve 1 (displayln (incrementar-contador)) ; Devuelve 2</pre>	<ul style="list-style-type: none"> <li>- La función incrementa-contador es impura porque modifica una variable global (contador).</li> <li>- El resultado depende del estado actual de contador, no solo de los inputs de la función.</li> </ul> <p>(En Racket, es posible declarar variables globales y modificarlas usando set!)</p>
--	--

## 5. Recursión en lugar de bucles

Los lenguajes funcionales suelen usar la recursión en lugar de los bucles (for, while). La recursión permite iterar de forma natural sobre estructuras como listas.

<pre>; Función recursiva para calcular el factorial (define (factorial n)   (if (zero? n)          ; Caso base: si n es 0, devuelve 1       1       (* n (factorial (- n 1))))) ; Caso recursivo: multiplica n por el factorial de n-1  ; Calcular el factorial de 5 (displayln (factorial 5)) ; Devuelve 120</pre>	<ul style="list-style-type: none"> <li>- La función factorial se llama a sí misma hasta que alcanza el caso base (n = 0).</li> <li>- Este enfoque es preferido en los lenguajes funcionales porque evita el uso de variables temporales o contadores.</li> </ul>
---	--

## 6. Composición de funciones

La composición de funciones consiste en combinar varias funciones para crear pipelines de transformaciones. Esto hace que el código sea modular y legible.

<pre>; Función para sumar 1 (define (sumar-1 x)   (+ x 1))  ; Función para multiplicar por 2 (define (multiplicar-2 x)   (* x 2))  ; Composición de las dos funciones (define (componer f g)   (lambda (x) (f (g x)))) ; Aplica primero g, luego f  (define doble-mas-uno (componer sumar-1 multiplicar-2))  ; Uso de la función compuesta (displayln (doble-mas-uno 3)) ; Devuelve 7 (3 * 2 + 1)</pre>	<ul style="list-style-type: none"> <li>- <b>componer</b> toma dos funciones f y g y devuelve una nueva función que aplica primero g y luego f.</li> <li>- La composición permite construir transformaciones complejas combinando funciones más simples.</li> </ul>
---	--

## 7. Evaluación perezosa contra evaluación estricta

La evaluación perezosa, o lazy evaluation, es una estrategia en la que las expresiones se calculan solo cuando es necesario, en lugar de hacerlo inmediatamente cuando se definen o se encuentran en el código. Esta no es una característica intrínseca de Racket.

La evaluación estricta, en cambio, utilizada por Racket (y en la mayoría de los lenguajes), se basa en los siguientes principios:

- Cada expresión se calcula inmediatamente cuando se encuentra en el código.
- Incluso los valores que podrían no ser utilizados nunca se calculan.
- Es más sencilla y predecible que la evaluación perezosa, porque no hay retrasos en los cálculos.

## 1.3. Historia de Racket

---

### Orígenes: Scheme y Lisp

Racket tiene sus raíces en Scheme, un dialecto de Lisp nacido en los años 70. Scheme fue diseñado por Guy L. Steele y Gerald Jay Sussman como un lenguaje minimalista y elegante:

- Scheme enfatizaba los conceptos de programación funcional y recursión.
- Era utilizado principalmente en el ámbito académico para enseñar conceptos de programación y realizar investigaciones sobre lenguajes.

Lisp en sí, el "lenguaje padre" de Scheme, se remonta a 1958 y es conocido por ser uno de los lenguajes de programación más antiguos, diseñado por John McCarthy.

### Desarrollo de Racket

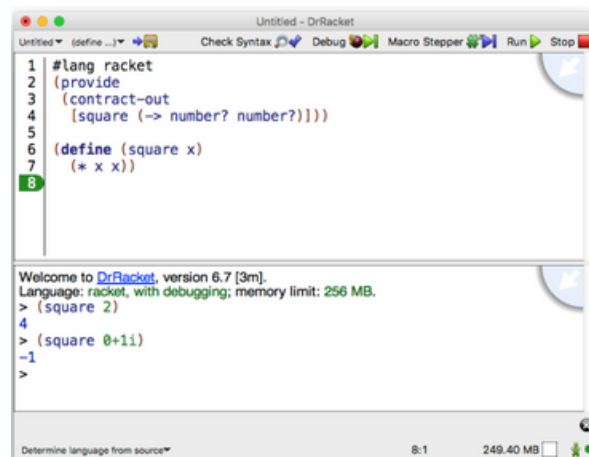
En los años 90, el profesor Matthias Felleisen y su equipo en el Northeastern University College of Computer and Information Science comenzaron a trabajar en Racket (entonces llamado PLT Scheme) como una extensión de Scheme. El objetivo era superar algunas limitaciones de Scheme y crear un lenguaje:

- Versátil: Adecuado para múltiples aplicaciones más allá de la pura programación funcional.
- Extensible: Capaz de soportar la creación de nuevos lenguajes específicos para dominios (DSL, Domain-Specific Languages).
- Didáctico: Ideal para la enseñanza, gracias a la simplicidad y claridad de su sintaxis.

## La evolución en un ecosistema completo

En 2010, PLT Scheme fue renombrado como Racket para reflejar su madurez como lenguaje independiente. Desde entonces, Racket ha crecido hasta convertirse en un ecosistema completo que incluye:

- **DrRacket:** Un entorno de desarrollo integrado (IDE) diseñado específicamente para Racket (un entorno de desarrollo integrado (IDE) es un programa que proporciona todo lo necesario para escribir, ejecutar y analizar el código en un solo lugar; en el caso de Racket, el IDE oficial es DrRacket, diseñado específicamente para simplificar el desarrollo con este lenguaje).



Es intuitivo para principiantes (ofrece una interfaz clara, con herramientas que ayudan a quienes comienzan a programar, como resaltado de sintaxis y guía de errores) y potente para desarrolladores experimentados (incluye funcionalidades avanzadas como depurador, perfilador y soporte para lenguajes personalizados).

- **Bibliotecas potentes:** Ofrece una amplia gama de bibliotecas predefinidas que simplifican el desarrollo de aplicaciones complejas. Estas bibliotecas garantizan las siguientes funcionalidades:
  - Creación y manipulación de gráficos e imágenes.
  - Diseño de interfaces gráficas para escritorio.
  - Herramientas para trabajar con protocolos de red y comunicación.
- **Soporte para DSL:** Racket permite crear Domain-Specific Languages (DSL), lenguajes diseñados para resolver problemas específicos en un ámbito determinado (p. ej., matemáticas, gráficos). Permite definir nuevas sintaxis y reglas, transformando a Racket en una "plataforma lingüística". Esto lo hace ideal para desarrolladores e investigadores que desean experimentar con nuevos paradigmas.
- **Herramientas para la investigación:** Gracias a su flexibilidad, es ampliamente utilizado en la investigación académica para desarrollar nuevos lenguajes y herramientas de programación.



## Racket hoy

Hoy, Racket es ampliamente utilizado en:

- **Educación:** Muchos cursos de programación funcional y lenguajes de programación adoptan Racket como lenguaje introductorio.
- **Investigación:** Es una herramienta esencial para experimentos sobre lenguajes y su extensibilidad gracias a su soporte para DSL.
- **Aplicaciones prácticas:** Aunque menos difundido que lenguajes como Python o Java, Racket se utiliza en contextos que requieren extrema flexibilidad, como la creación de software personalizado y herramientas para necesidades empresariales específicas.

## 1.4. Soporte para DSL

---

Un Domain-Specific Language (DSL) es un lenguaje de programación diseñado para un dominio o contexto específico, como el cálculo matemático, el control de procesos industriales o la generación de gráficos.

Racket ofrece herramientas avanzadas para crear nuevos lenguajes personalizados (DSL) a partir de su infraestructura. Esto es lo que lo convierte en una verdadera "plataforma lingüística": no solo un lenguaje de programación, sino un sistema para construir lenguajes.

Racket permite de:

Definir nuevas reglas sintácticas y semánticas mediante macros:

Racket te permite personalizar la sintaxis y el significado (semántica) del código, añadiendo nuevas reglas o modificando las existentes. Esto es útil para adaptar el lenguaje a las necesidades específicas de un dominio.

Uso de macros:

Cuando escribes una macro, estás definiendo una regla que "expande" un fragmento de código en otro.

Las macros te permiten inventar nuevas reglas o construcciones sintácticas, como un nuevo tipo de bucle o una función personalizada, sin cambiar el lenguaje base.

Esto permite, por tanto, crear nuevos lenguajes desde cero, utilizando como base Racket y modificando la sintaxis y la semántica mediante el uso de macros.

# SINTAXIS BASADA EN S-EXPRESIONES

## 2.1. Definición y Características

---

Las expresiones simbólicas, comúnmente conocidas como S-expresiones, son la forma de representar datos y código del lenguaje Racket y de la familia Lisp en general. Se estructuran a partir de listas de símbolos y operadores con notación prefija a diferencia de lenguajes más tradicionales que usan una sintaxis basada en palabras clave y operadores.

En su forma más básica, una S-expresión puede ser un número, una cadena de caracteres, un símbolo, una función o una lista. Las listas se construyen mediante la inclusión de un conjunto de elementos entre paréntesis, donde los elementos pueden ser otros símbolos, valores o incluso otras listas.

### **Uniformidad**

Las S-expresiones presentan una estructura uniforme, ya que cualquier expresión en Racket se puede representar como una lista. Esto facilita la manipulación del código pues comparte la misma representación que los datos.

### **Simplicidad**

Permiten que la sintaxis del lenguaje sea extremadamente sencilla, eliminando la necesidad de reglas de precedencia, de asociación o estructurales.

### **Recursividad**

Las listas en Racket son recursivas por naturaleza, lo que significa que pueden contener otras listas dentro de ellas. Esta propiedad permite representar estructuras complejas de datos y operaciones sin necesidad de un formato más rígido.

### **Fácil análisis sintáctico**

La notación prefija simplifica el diseño de parsers, ya que las expresiones se representan directamente como árboles sintácticos.

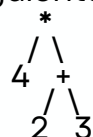
## 2.2. Cómo se utilizan

### Ejemplos básicos

Una expresión puede ser un número, cadena de caracteres o la aplicación de una función predefinida o aritmética a un número de argumentos que pueden ser determinados o no, como vemos a continuación.

```
> 12
> -25
> 3.14
> "Hola mundo"
> (min 0.01 -5 20 1/2)
  -5
> (* (+ 1 2 3 4) (- 5 6) (/ 7 8 9))
  -0.97222
```

Para ver como se evalúa consideremos por ejemplo, la expresión  $(* 4 (+ 2 3))$  en notación prefija que corresponde al siguiente árbol sintáctico:



Este árbol se evalúa directamente en orden descendente (postorden), reflejando el modelo de evaluación funcional, donde las operaciones se aplican directamente a los operandos, y el resultado reemplaza a la expresión en el contexto de evaluación. Resultando  $(* 4 (+ 2 3)) = 4 * (2 + 3) = 20$ .

### Definiciones

Existen dos tipos de definiciones una en la que se le asigna a un identificador un valor, y otra para funciones, donde se especifica el nombre de la función y sus parámetros, y el cuerpo de la función.

```
(define x 2)
(define y (+x 1))
(define (suma' x y) (+ x (square y))) ;(def (name params) (body))
> (suma' x y)
11
```

### Funciones Lambda

En Racket, como en otros lenguajes funcionales, las funciones son tratadas como valores. Esto significa que pueden ser almacenadas en variables, pasadas como argumentos a otras funciones, devueltas como resultados, e incluso creadas dinámicamente durante la ejecución del programa. Esto permite que una expresión sea evaluada con argumentos específicos en tiempo de ejecución, proporcionando una gran flexibilidad y expresividad en la programación funcional.

```
(define (traducir s)
  (cond [(equal? s 'sumar) +]
        [(equal? s 'restar) -]))

>((traducir 'sumar) 3 4)
7
```

Para crear funciones dinámicamente emplearemos la forma *lambda*. Lambda crea una expresión que da como valor una función anónima, a la que le podremos asociar un nombre. Luego tenemos dos formas de definir la función suma por ejemplo.

```
(define (suma x y) (+ x y))

(define suma (lambda (x y) (+ x y)))
```

Estas dos formas son equivalentes pero es más la primera es azúcar sintáctico para la segunda (antes de ejecutarse el programa se traduce a la segunda forma). Luego todas las definiciones le darán nombre a valores, en concreto las definiciones de funciones le dan nombre a expresiones lambda.

Una expresión lambda no siempre va a ser un valor como en el ejemplo anterior, habrá veces que requiera de una ejecución posterior.

```
(define (crear-sumador n)
  (lambda (x) (+ x n)))

>(crear-sumador 2)
#<procedure>

>((crear-sumador 2) 3)
5
```

En este caso *(crear-sumador 2)* es un procedimiento asociado a la función lambda pero donde *n*, que es su variable libre, tiene valor dos; es decir está asociado a *(lambda (x) (+ x 2))*. A esto se le llama **clausura** formada por la expresión lambda y las asociaciones o *bindings* de todas sus variables libres en el momento de creación.

### Funciones de orden superior

Una función de orden superior es una función donde uno de sus argumentos es otra función, o que el resultado es una función. La función *map* recibe como argumentos una función y una lista y aplica a cada elemento de la lista la función, luego es una función de orden superior.

```
>(map (lambda (x) (* x x)) '(1 2 3 4))
'(1 4 9 16)
>(map odd? '(1 2 3 4))
'(#t #f #t #f)
```

Una implementación básica de la función map sería la siguiente.

```
(define (map' func lista)
  (cond [(empty? lista) empty]
        [else (cons (func (first lista)) (map' func (rest lista))))])
```

Esta función además de otras como la usada para sumar los elementos de una lista se pueden generalizar a través del caso base (lista vacía) y la función usada para combinar el primer elemento de la lista con el resto de la llamada recursiva. Para ello se emplean las funciones predefinidas *foldr* y *foldl*.

```
(define (foldr comb caso_base lst)
  (cond
    [(empty? lista) caso_base]
    [else (comb (first lst) (foldr comb caso_base (rest lst)))]))
```

También es posible definir funciones que tomen un numero arbitrario de argumentos, o que tengan argumentos por defecto.

```
(define sqr-lista
  (lambda args (map sqr args)))

(define resta
  (lambda (x [y 1]) (- x y)))

>(sqr-lista 1 2 3)
'(1 4 9)
>(resta 4 2)
2
>resta 3
2
```

## Variables Locales

Las variables locales en Racket se pueden expresar usando expresiones lambda pero esto es poco intuitivo así que se introduce *let* que es azúcar sintáctico para la expresión.

```
>((lambda (x y) (+ (* x x) (*y y))) (+ 1 2) (- 4 1))
18

>(let ([x (+ 1 2)]
       [y (- 4 1)])
      (+ (* x x) (* y y)))
18
```

Asimismo cuando queremos usar una variable local ya definida para definir otra podemos anidar lets o equivalentemente usar *let\**.

```
>(let* ([x (+ 1 2)]
        [y (- 4 x)])
      (+ x y))
4
```

Si queremos que la variable local sea una función deberemos usar *lambda* para definirla, aunque Racket también permite definiciones internas en el cuerpo de las expresiones lambdas o de las funciones definidas. Además si queremos usarla de forma recursiva deberemos usar *letrec*.

```
> (letrec ([even? (lambda (n) (or (zero? n)
                                  (odd? (sub1 n))))]
           [odd? (lambda (n) (and (not (zero? n))
                                   (even? (sub1 n))))])
  (even? 12))
#t
```

# 03

## SISTEMA DE TIPOS

### 3.1. Tipos Dinámicos

---

Racket se caracteriza por tener un tipo de datos dinámico, es decir, no se requiere inicialización del tipo sino que se establece en tiempo de ejecución, por lo que el tipo está asociado al valor de la variable en ese momento.

```
(define x 5)
(define x "Hola")
```

Por tanto hay funciones que pueden ejecutarse con tipos de datos distintos cada vez y pueden también producir valores de distintos tipos, por ejemplo la función *cons* que crea una lista a partir de dos elementos, la cabeza y la cola, resultando una lista que puede contener tipos distintos.

```
> (cons true (cons "blue" 4))
(#t "blue" 4)
```

Cuando existe una operación con un valor de tipo incorrecto, se obtiene un error de tipo.

```
(define x 42)          ; x tiene un valor numérico
(define y "Hola")      ; y tiene un valor de tipo cadena

>(+ x 1)                ; Correcto: suma un número
43

> (+ x y)               ; Error en tiempo de ejecución: suma num y cadena.
Error
```

Como los errores de tipo no se detectan hasta que se intenta realizar una operación inválida, los programas son más flexibles, pero también se introduce el riesgo de que los errores pasen desapercibidos hasta su ejecución.

## Contratos

Para mitigar los riesgos asociados con el tipado dinámico, Racket ofrece los contratos. Los contratos permiten definir restricciones explícitas sobre los valores que una función puede aceptar o devolver y se verifican en tiempo de ejecución.

```
> (define/contract (maybe-invert i b)
  (-> integer? boolean? integer?)
  (if b (- i) i))
> (maybe-invert 1 #t)
-1
> (maybe-invert #f 1)
maybe-invert: contract violation
  expected: integer?
  given: #f
  in: the 1st argument of
      (-> integer? boolean? integer?)
  contract from: (function maybe-invert)
  blaming: top-level
    (assuming the contract is correct)
  at: eval:2:0
```

Los contratos no sustituyen a un sistema de tipos estáticos sino que ofrecen mayor flexibilidad y proporcionan un nivel adicional de seguridad en tiempo de ejecución. Esto se debe a que los contratos son dinámicos y semánticos, lo que garantiza propiedades de los valores que, en muchos casos, no pueden formalizarse mediante un sistema de tipos estáticos.

## 3.2. Tipos Estáticos

---

A pesar de que Racket emplea un sistema de tipos dinámicos, su ecosistema incluye herramientas que permiten trabajar con sistema de tipos estáticos. Typed Racket, una extensión de Racket, introduce este paradigma, donde los tipos de las variables y expresiones se determinan y verifican durante la fase de compilación, antes de que el programa se ejecute, ofreciendo así una mayor robustez al programa y minimizando el número de errores en tiempo de ejecución.

El compilador de Typed Racket se asegura de que las operaciones son válidas y previene inconsistencias antes de la ejecución del programa, detectando errores con los tipos.

El tipado estático ofrece también un rendimiento optimizado, ya que el compilador generará un código más eficiente; una detección temprana de errores y documentación implícita que ayuda a la comprensión del código.

## Typed Racket

Typed Racket permite especificar tipos para variables, funciones y estructuras sin perder la expresividad propia del lenguaje mediante anotaciones explícitas que son verificadas por el compilador. Su uso se habilita mediante la declaración `#lang typed/racket`.

Los tipos básicos que se suelen emplear son *Integer*, *Real*, *String* y *Boolean*. Las listas se denotan como *Listof* y las tuplas como *Pair*. Además existe el tipo *Any* que es un tipo genérico.

En las definiciones de variables o de funciones el tipo se puede expresar de distintas formas como se puede observar en los ejemplos siguientes.

```
(: x Number)
(define x 7)

(define x : Number 7)

(: mas1 (-> Number Number))
(define (mas1 z) (+ z 1))

(define (mas1 [z : Number]) : Number (+ z 1))

(: id (All (T) (-> T T))) ; T es un tipo genérico
(define (id x) x)
```

Se permite también la definición de estructuras para su posterior empleo en funciones como para calcular la distancia entre dos puntos en un plano:

```
#lang typed/racket
(struct pt ([x : Real] [y : Real]))
(: distance (-> pt pt Real))
(define (distance p1 p2)
  (sqrt (+ (sqr (- (pt-x p2) (pt-x p1)))
           (sqr (- (pt-y p2) (pt-y p1))))))

> (distance (pt 0 0) (pt 3.1415 2.7172))
- : Real
4.153576541969583
> distance
- : (-> pt pt Real)
#<procedure:distance>
> (:print-type string-length)
(-> String Index)
```

Nótese que al ejecutar una función antes del resultado se devuelve también su tipo, además si ejecutamos únicamente el nombre de la función se imprime en pantalla el tipo de la función y su valor.



Otro elemento muy útil es la unión de tipos empleado sobre todo en estructuras de datos, como puede ser en el caso de los árboles binarios que pueden ser un nodo que tendrán árbol derecho e izquierdo o una hoja que contiene un único valor. Permitiendo también la definición de estructuras recursivas.

```
#lang typed/racket
(define-type Tree (U leaf node))
(struct leaf ([val : Number] )
(struct node ([left : Tree] [right : Tree]))
```

También es posible anotar expresiones mediante *ann* con la expresión como primer argumento y el tipo esperado como segundo, empleado para validar que la expresión cumple con el tipo esperado.

```
(ann (+ 1 2) Number) ;Correcto
(ann "Hola" Number) ;Error type mismatch
```

### 3.3. Comparación

Características	Tipado Dinámico		Tipado Estático
	General	Contratos	
<b>Verificación de Tipos</b>	En tiempo de ejecución	En tiempo de ejecución	En tiempo de compilación
<b>Flexibilidad</b>	Alta, no requiere especificación de los datos	Alta, permite dar propiedades sobre los datos	Menor, requiere anotaciones explícitas y claras.
<b>Seguridad</b>	Baja, pueden aparecer errores en la ejecución	Alta, asegura props. dinámicas de los valores	Alta, los errores se detectan antes de ejecutar
<b>Expresividad</b>	Eficaz para escenarios simples y rápidos	Para validar comportamientos complejos	Moderada, formalización detallada
<b>Rendimiento</b>	Bajo (verificaciones en tiempo de ejecución)	Similar al dinámico	Alto (las verificaciones permiten optimización)
<b>Uso en Racket</b>	Predeterminado	<i>require/contract</i>	<i>#lang typed/racket</i>

## 3.4. Tipado Gradual

---

Racket a través de Typed Racket soporta el tipado gradual, un enfoque que combina los beneficios del sistema de tipos dinámico y del estático. Esto permite decidir de forma flexible cuándo y dónde especificar tipos en los programas, haciendo posible la coexistencia de código dinámico y estático en un mismo proyecto. Es decir, se permite escribir partes de un programa con tipos explícitos verificados en tiempo de compilación, mientras que otras se mantienen sin tipar y se evalúan dinámicamente en tiempo de ejecución. Esto además facilita la migración progresiva de un sistema dinámico a uno estático, por lo que es especialmente útil en proyectos que requieren un desarrollo rápido inicial y necesitan robustez y escalabilidad a largo plazo.

### Características

- Interoperabilidad: los módulos y funciones con tipado estático pueden interactuar con los dinámicos de manera segura. Para ello se realizan verificaciones dinámicas adicionales en tiempo de ejecución, garantizando así que no haya errores de tipo.
- Adaptación Progresiva: se puede añadir los tipos a un programa de manera incremental según sea necesario, aumentando así la detección temprana de errores y por tanto la seguridad del programa.

### Limitaciones

- Sobrecarga en tiempo de ejecución debido a las verificaciones dinámicas entre módulos.
- Complejidad adicional ya que combina dos sistemas de tipos diferentes.

### Implementación

El tipo `Any` se emplea como puente entre códigos de distintos tipos, permitiendo así valores sin especificación explícita de tipo. Además se emplean los contratos para verificar que se cumple con las expectativas de los tipos declarados.

Un ejemplo de verificación en tiempo de ejecución se daría si tenemos un código con tipos estáticos que espera un valor de tipo `T` y recibe un valor `v` de tipo dinámico. Así que se comprobaría que `v` es de tipo `T` en tiempo de ejecución empleando por ejemplo el siguiente código.

```
(if (T? v) v (error 'bad-type))
```

# 04

## MÓDULOS, MACROS Y SU USO PARA LA CREACIÓN DE DSLs

### 4.1. Módulos en Racket

---

#### ¿Qué son y cómo funcionan?

Los módulos en Racket son fundamentales para la organización y el manejo de dependencias en el desarrollo de software. Este sistema permite dividir el código en componentes independientes y reutilizables, facilitando la colaboración en proyectos grandes, la gestión de librerías, y el control de visibilidad de los elementos del código.

#### Sintaxis de un módulo

La forma extendida de una declaración de módulo, que funciona tanto en un REPL como en un archivo, es:

```
(module name-id initial-module-path  
      decl ...)
```

donde **name-id** es un nombre para el módulo, **initial-module-path** es una importación inicial, y cada **decl** es una importación, exportación, definición o expresión. En el caso de un archivo, **name-id** suele coincidir con el nombre del archivo que lo contiene, sin su ruta de directorio ni extensión de archivo, pero se ignora cuando el módulo se requiere a través de la ruta de su archivo.

#### Require como import

La forma `require` importa desde otro módulo. Una forma `require` puede aparecer dentro de un módulo; en ese caso, introduce enlaces desde el módulo especificado en el módulo importador. Una forma `require` también puede aparecer en el nivel superior; en este caso, no solo importa los enlaces, sino que instancia el módulo especificado, es decir, evalúa las definiciones y expresiones del cuerpo del módulo especificado si aún no han sido evaluadas.


## Provide como export

Una forma provide solo puede aparecer a nivel de módulo (es decir, en el cuerpo inmediato de un módulo). Especificar múltiples provide-specs en un único provide es exactamente lo mismo que usar varios provide, cada uno con un único provide-spec.

Cada identificador puede exportarse, como máximo, una sola vez desde un módulo en todos los provide dentro del módulo. Más precisamente, el nombre externo de cada exportación debe ser único; el mismo enlace interno puede exportarse varias veces con diferentes nombres externos.

## Ejemplo “cake.rkt”

En este caso vamos a crear un módulo que simule el dibujo de una tarta dado un número de velas y lo exportaremos usando (**provide print-cake**) como podemos ver.

cake.rkt (define ...)	random-cake.rkt (define ...)
<pre>1 #lang racket 2 3 (provide print-cake) 4 5 ; draws a cake with n candles 6 (define (print-cake n) 7   (show "  ~a  " n #\.) 8   (show " .-~a-." n #\ ) 9   (show "   ~a   " n #\space) 10  (show "----~a----" n #\~)) 11 12 (define (show fmt n ch) 13   (printf fmt (make-string n ch)) 14   (newline)) 15</pre>	<pre>1 #lang racket 2 3 (require "cake.rkt") 4 5 (print-cake (random 30)) 6</pre> <p>Welcome to <a href="#">DrRacket</a>, version 8.14 [cs]. Language: <a href="#">racket</a>, with debugging; memory limit: 128 MB.</p> 

Veamos como funciona el módulo creando otro documento distinto llamado “**random-cake.rkt**”

El ejemplo “**cake.rkt**” y “**random-cake.rkt**” demuestra la forma más común de organizar un programa en módulos: colocar todos los archivos de módulo en un solo directorio (quizás con subdirectorios) y hacer que los módulos se referencien entre sí a través de rutas relativas. Un directorio de módulos puede actuar como un proyecto, ya que puede moverse en el sistema de archivos o copiarse a otras máquinas, y las rutas relativas mantienen las conexiones entre los módulos.

## 4.2. Macros en Racket

### ¿Qué son y cómo funcionan?

Las macros en Racket son una de sus características más poderosas y distintivas, y se basan en la capacidad de manipular y expandir el código antes de que se ejecute. A diferencia de otros lenguajes de programación, en los que las macros son más limitadas, en Racket, las macros nos permiten crear nuevas construcciones y hasta definir nuestras propias reglas sintácticas.

Una macro es una forma sintáctica con un transformador asociado que expande la forma original en formas existentes. Dicho de otro modo, una macro es una extensión del compilador de Racket. La mayoría de las formas sintácticas de racket/base y racket son, de hecho, macros que se expanden en un conjunto reducido de construcciones básicas.

### Sistema de Macros Higiénico en Racket

Una de las características distintivas de las macros en Racket es su "higiene", que garantiza que las macros respeten el alcance léxico y eviten conflictos entre identificadores. La higiene en macros significa que los nombres de variables definidos dentro de una macro no entran en conflicto con otros nombres del programa, evitando errores de colisión de nombres.

### Ejemplo práctico

En Racket, la función `print` está construida para recibir un único parámetro para mostrar por pantalla. Creamos una macro para poder mostrar más de un elemento usando un único `print`.

```
1 | #lang racket
2 | (require (for-syntax syntax/parse))
3 |
4 | (define-syntax (print form)
5 |   (syntax-parse form
6 |     ((print arg:expr ...)
7 |      #`(begin
8 |        (display arg) ...
9 |        (newline)))))
```

Importamos `(for-syntax syntax/parse)` para poder parsear el código y diferenciar sus partes y usamos `define-syntax` para definir la macro que llamaremos `print form`.

Definimos la forma que tendrá nuestra línea de código indicando sus tipos acompañado con `...` para indicar multiplicidad.

```
Welcome to DrRacket, version 8.14 [cs].
Language: racket, with debugging; memory limit: 128 MB.
> (print "1+1=" (+ 1 1))
1+1=2
>
```

Usamos el `#`` ( para comenzar la “traducción” del código de la siguiente manera: lo interpretaremos como varios `display`, uno por cada argumento que haya. De esta manera hemos conseguido que cuando compilemos el código la instrucción se interpretará gracias a la nueva macro como varios `display`.

## 4.3. Racket como Lenguaje orientado a lenguajes

### ¿Qué son los Lenguajes Específicos del Dominio (DSL)?

Un Lenguaje Específico del Dominio (DSL) es un tipo de lenguaje de programación diseñado con un nivel de abstracción alto y optimizado para resolver problemas en un ámbito concreto. Estos lenguajes utilizan las reglas y conceptos característicos de su campo o dominio, facilitando el trabajo en áreas especializadas.

### ¿En qué se diferencian los Lenguajes Específicos del Dominio de los lenguajes de propósito general?

Los DSL suelen ser menos complejos que los lenguajes de propósito general, como Java, C o Ruby. En general, un DSL se desarrolla en colaboración estrecha con expertos en el área de aplicación, asegurando que el lenguaje sea accesible y efectivo en ese contexto específico.

### Racket como Meta-lenguaje para DSLs

Racket es un "meta-lenguaje" que se destaca por su capacidad para crear otros lenguajes de programación a través de un sistema de macros avanzado y herramientas específicas. Gracias a su diseño modular y a su sistema de macros higiénicas, Racket facilita la construcción de DSLs, proporcionando un entorno flexible para crear sintaxis y semántica personalizadas.

### Setup Básico para Crear un DSL en Racket: Expander y Reader

Para crear un DSL en Racket, es esencial entender dos componentes principales: el Reader y el Expander.

El reader convierte el código fuente del nuevo lenguaje en formas con paréntesis al estilo de Racket, llamadas S-expresiones. El reader debe proporcionar una función llamada `read-syntax`. Racket pasa dos argumentos a esta función: la ruta al archivo fuente y un puerto de entrada para leer el código fuente. Después de convertir el código fuente a S-expresiones, la función debe devolver código de Racket que describa un módulo (empaquetado como un objeto de sintaxis). En el archivo fuente, Racket reemplaza el código fuente con este código de módulo.

El expander determina cómo las S-expresiones generadas por el reader corresponden a expresiones reales de Racket, asegurándose de que cada identificador tenga una vinculación. El expander debe proporcionar una macro llamada `#%module-begin`. Racket toma el código dentro de la expresión de módulo generada por el reader y lo pasa a esta macro. Esta macro debe devolver nuevo código de Racket, nuevamente empaquetado como un objeto de sintaxis. Racket reemplaza la expresión de módulo generada por el reader con este nuevo código.

Una vez que el expander termina su trabajo, el nuevo lenguaje ha sido traducido a código Racket ordinario. Se evalúa como un programa de Racket para producir un resultado.

### Ejemplo práctico: creación del lenguaje GOTO

En nuestro ejemplo utilizaremos un lenguaje GOTO bastante simple donde destaca su uso de etiquetas e instrucciones GOTO para viajar entre ellas. Cabe destacar que todas las variables usadas en este lenguaje son de tipo entero y además todas las instrucciones comenzarán con las mencionadas etiquetas. Todas las instrucciones se ejecutan sucesivamente una tras otra teniendo en cuenta los posibles saltos de código.

Las instrucciones que nos podemos encontrar en este lenguaje son:

- *Asignaciones*: almacenan en la variable indicada la expresión correspondiente, por ejemplo,  $A = 1$  o  $B = C + 2$
- *Print*: muestra por pantalla el valor de la variable indicada, por ejemplo, `PRINT A`
- *Return*: finaliza el programa
- *GOTO*: al ejecutar esta línea seguiremos ejecutando el código desde la línea indicada, ejemplo `GOTO 10`
- *GOSUB*: al ejecutar esta línea seguiremos ejecutando el código desde la línea indicada hasta encontrar un return que regresará al punto de partida.
- *Condicional*: la única instrucción condicional usada en este lenguaje es `if` que funciona de manera habitual y sigue la siguiente sintaxis, `IF expr THEN expr ELSE expr`, como por ejemplo `IF X < 0 THEN PRINT "DONE" ELSE GOTO 20`

Ejemplo de programa:

```
10 A = 1
20 PRINT A
30 A = A + 1
40 IF A < 100 THEN GOTO 20 ELSE PRINT "DONE"
```

Mostrará por pantalla cada valor de A hasta llegar a 99 y "DONE" para terminar. Ahora sí comencemos con la creación del lenguaje GOTO implementando Racket.



## Expander (archivo goto-expander.rkt):

En la siguiente imagen podemos ver los fragmentos más importantes de código explicados. Todos los archivos usados se enviarán en la entrega del proyecto.

```
#lang racket
(provide goto-language print goto := gosub return) → Exportamos
(require (for-syntax syntax/parse → Para parsear el código
         racket/syntax → Partimos de la sintaxis de Racket
         syntax/id-set)) → Lo usaremos para el
                           almacenamiento de las variables
Importamos

Definimos la sintaxis con
el nombre goto-language

(define-syntax (goto-language form)
  (syntax-parse form
    ((goto-language (line-number:integer command:expr) ...)
     (define id-set (mutable-free-id-set))
     (for-each (lambda (command)
                 (collect-variables command id-set))
              (syntax->list #'(command ...)))
     #`(begin
          #,@(map (lambda (variable)
                    #`(define #,variable #f))
                  (free-id-set->list id-set))
          #,@(map (lambda (line-number next-line-number command)
                    (define name (make-line-name #'goto-language line-number))
                    (define call-next-line → Función para que ejecute todas las líneas
                                             sucesivamente
                    (if next-line-number
                        #`(#,(make-line-name #'goto-language next-line-number))
                        #`(void))) → Si no hay más líneas usamos void (no hace nada)
                    #`(define (#,name)
                          #,(translate-command #'goto-language command call-next-line)))
                  (syntax->list #'(line-number ...))
                  (append (cdr (syntax->list #'(line-number ...))) '(#f))
                  (syntax->list #'(command ...)))))))

Traducimos a lenguaje Racket

Para traducir la funcionalidad de los comandos de GOTO
(define-for-syntax (translate-command context command call-next-line)
  (syntax-parse command
    #:literals (goto := if gosub return)
    ((goto line-number:integer)
     #`(#,(make-line-name context #'line-number)))
    ((gosub line-number:integer)
     #`(begin
          (#,(make-line-name context #'line-number))
          #,call-next-line))
    ((return) (void)) → Finaliza el programa
    ((:= variable:id rhs:expr)
     #`(begin
          (set! variable rhs)
          #,call-next-line))
    ((if test:expr then:expr else:expr)
     #`(if test
          #,(translate-command context #'then call-next-line)
          #,(translate-command context #'else call-next-line)))
    (_
     #`(begin #,command
              #,call-next-line))))

Ejecuta la línea citada en GOTO _
Ejecuta la línea citada en GOSUB _ y regresa al punto de partida cuando encuentra un return
Almacena en el set la variable y el valor indicado en la asignación
Funcionalidad común del condicional "if"
```



### Reader (archivo goto-reader.rkt):

Como hemos visto, la base para crear un lenguaje de programación propio reside en el expander, mientras que en el reader nos encontraremos un enfoque más dirigido a la sintaxis y menos a la semántica con la definición de tokens, módulos y errores a la hora de parsear el código. Es por eso que no nos centraremos tanto en este archivo.

Comenzaremos indicando que el código de este documento se escribirá en lenguaje racket, escribiendo así `#lang racket`

Luego usaremos la segunda línea de código para permitir que goto-read-syntax esté disponible para otros módulos bajo el nombre de read-syntax.

```
#lang racket
(provide (rename-out (goto-read-syntax read-syntax)))
(require syntax/readerr)

(define (goto-read-syntax src in)
  (datum->syntax
   #f
   `(module goto-language racket
      (require "goto-expander.rkt")
      (goto-language
       ,@ (parse-program src in))))))
```

Usaremos syntax/readerr para manejar errores de sintaxis.

Seguidamente encontramos la función principal del reader que procesa el código fuente cuando se carga este lenguaje. Vemos que convierte los datos en un objeto de sintaxis. Luego crea el módulo goto-language basado en racket usando el archivo mencionado anteriormente “goto-expander.rkt” que contiene las transformaciones específicas para construir goto-language. Por último, parse-program convierte la entrada en expresiones de Racket apropiadas para goto-language.

### Demo (archivo goto-demo.rkt):

Vamos a implementar la siguiente función en el lenguaje GOTO. Las variables usadas son: Z como variable auxiliar, Y como salida y X como entrada. La funcionalidad es sencilla:

1. Almacenamos en Z el valor de  $Y^2$
2. Comprobamos si el valor obtenido es menor o igual que X
  - 2.1. Si efectivamente es menor, nos vamos a C, que aumenta una unidad en nuestra variable de salida y una vez aumentada volvemos al inicio del código
  - 2.2. Si Z resulta ser mayor que X entonces debemos retroceder en nuestra variable de salida una unidad para así obtener el máximo  $n^2$  que cumple la condición de ser menor que X.

```

f(x) = max { n : n^2 ≤ x }
[A] Z ← Y^2
    IF Z ≤ X GOTO C ELSE GOTO B
[B] Y ← Y - 1
    PRINT Y
    RETURN
[C] Y ← Y + 1
    GOTO A

```

Escribámos este código en Racket usando el lenguaje creado, para ello debemos escribir `#lang reader "goto-reader.rkt"`

En efecto, el código funciona y si nuestra variable de entrada vale 10, obtenemos que el mayor número cuyo cuadrado es menor de 10 es 3 como indica en la consola.

```

1 | #lang reader "goto-reader.rkt"
2 | 10 X = 10
3 | 20 Z = 0
4 | 30 Y = 0
5 | 40 Z = Y * Y
6 | 50 IF Z <= X THEN GOTO 90 ELSE GOTO 60
7 | 60 Y = Y - 1
8 | 70 PRINT Y
9 | 80 RETURN
10 | 90 Y = Y + 1
11 | 100 GOTO 40

```

```

Welcome to DrRacket, version 8.14 [cs].
Language: reader "goto-reader.rkt", with debugging; memory limit: 128 MB.
> (line-10)
3
>

```

# CASOS DE USO Y EJEMPLOS

## 5.1 Racket en la docencia

---

Racket tiene un marcado carácter educativo, que se está utilizando tanto en nivel medio como en el ámbito de la educación superior. El lenguaje le ofrece un entorno muy adecuado para la enseñanza de los fundamentos de programación, así como para la enseñanza de temas más avanzados en ciencias de la computación.

Los principales enfoques de este lenguaje en la docencia son los siguientes:

- **Enseñanza de las bases de la programación:** La estrategia inicial de Racket ha permitido a los estudiantes aprender conceptos básicos de programación mediante un lenguaje simple, a diferencia de otros lenguajes más extensos e inflexibles. Su sintaxis simple, pero funcional, da la oportunidad de enseñar a los estudiantes el paradigma de la programación estructurada y funcional sin distraerles con detalles de otros lenguajes más complejos. Sus características principales que lo destacan de otros lenguajes en la enseñanza son:
  - **Sintaxis minimalista.** Su escritura sencilla permite el enfoque en los conceptos importantes.
  - **Evaluación de expresiones.** Este lenguaje permite experimentar con la evaluación de expresiones, un concepto básico de la programación funcional.
  - **Funciones como parámetros.** Los estudiantes pueden aprender fácilmente a manejar funciones como parámetros, lo cual es crucial para entender la programación funcional
- **Resolución de problemas:** Este lenguaje favorece el pensamiento lógico y el pensamiento estructurado al centrarse en la solución de problemas por medio de funciones. Los estudiantes pueden descomponer problemas complejos en funciones, fomentando así la mejora del razonamiento y su capacidad analítica.

- **Modelado de conceptos abstractos:** Al ser un lenguaje extensible permite la creación de nuevos lenguajes dentro del entorno, esto hace que se puedan modelar y estudiar lenguajes de programación, experimentar con definiciones propias de semántica y sintaxis...
- **Uso en el desarrollo de algoritmos:** El lenguaje permite implementar algoritmos de manera clara y sin la interferencia de aspectos técnicos más complejos de otros lenguajes, facilitando así el aprendizaje de estructuras de datos, algoritmos y técnicas computacionales de manera eficiente.

## 5.2 Ventajas de su uso

---

- **Herramientas integradas:** Una de las herramientas integradas en este lenguaje es **Dr.Racket**, que es un IDE diseñado específicamente para la enseñanza. Este incluye algunas facilidades como depuración interactiva, syntax highlighting, y la modificación y creación de nuevos lenguajes de programación.
- **Enfoque en la comprensión conceptual:** La capacidad de este lenguaje para modelar conceptos como la recursión y las estructuras de datos lo convierte en un excelente medio para la enseñanza de la teoría de la computación. Esto es gracias a las facilidades en la implementación de conceptos abstractos antes mencionadas.
- **Extensibilidad:** Al ser un lenguaje altamente extensible, permite que se pueda adaptar a las necesidades de un curso o una materia. Los profesores podrían crear dialectos personalizados del lenguaje de forma que se ajuste a los objetivos educativos.
- **Soporte de la comunidad:** Este lenguaje tiene una comunidad activa de profesores y desarrolladores que comparten siempre recursos, materiales educativos y ejemplos de código. Esto ayuda a crear contenido educativo de calidad y a resolver problemas comunes que los docentes puedan tener.

## 5.3 Ejemplos de uso en la docencia

---

- **Curso introductorio de programación:** Uno de los casos más comunes del uso de Racket es el aprendizaje de los fundamentos de la programación funcional en cursos introductorios. Primero, los estudiantes aprenden a escribir funciones muy sencillas y, gradualmente, pasan a la recursión, las listas y los árboles, con Racket como lenguaje de instrucción.
- **Modelado de lenguajes de programación:** En cursos avanzados de teoría de lenguajes de programación, Racket es una herramienta útil para enseñar cómo definir y analizar lenguajes. Los estudiantes pueden crear sus propios dialectos de Racket, lo que les permite explorar la semántica y la interpretación de código.
- **Proyectos de programación funcional:** En proyectos, los estudiantes pueden implementar algoritmos funcionales complejos, como el manejo de estructuras de datos inmutables o la implementación de funciones recursivas. Esto les permite asentar sus conocimientos de programación funcional de manera práctica.

# BIBLIOGRAFÍA

<https://cs.uwaterloo.ca/~plragde/flaneries/TYR/>

<https://docs.racket-lang.org>

<https://lenguajes.gitlab.io/20191/notas/ldp20191na1.pdf>

[https://media.ccc.de/v/rc3-257534-all\\_programming\\_language\\_suck\\_just\\_build\\_your\\_own\\_language\\_oriented\\_programming\\_with\\_racket#t=3120](https://media.ccc.de/v/rc3-257534-all_programming_language_suck_just_build_your_own_language_oriented_programming_with_racket#t=3120)

<https://docs.racket-lang.org/guide/macros.html>

<https://docs.racket-lang.org/guide/modules.html>