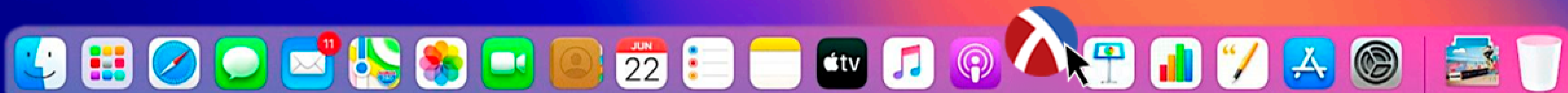




Teoria_de_los_Lenguajes_de_Programacion.rkt - DrRacket

```
#lang racket  
(define título  
  "Programación funcional - Racket")  
  
(define participantes  
  (list "María Peinado" "Yolanda  
Sarmiento" "Pedro Sánchez" "Andrea  
Lazzaretto"))  
;; Curso 24-25
```



✓ ÍNDICE


✓ 01_Introducción

- ≡ 1.1_Filosofia Detras.txt
- ≡ 1.2_Lenguajes funcionales.txt
- 1.3_Histora
- 1.4_Soporte para DSL.txt


✓ 02_Sintaxis basada en S-expresiones

- ≡ 2.1_Definicion y características.txt
- ≡ 2.2_Cómo se utilizan.txt

✓ 03_Sistema de tipos

- ≡ 3.1_Tipos dinámicos.txt
- ≡ 3.2_Tipos estáticos.txt
- ① 3.3_Comparación.md
-  3.4_Tipado Gradual.rkt

✓ 04_Módulos, Macros y su uso para la creación de DSL

- ≡ 4.1_Módulos en Racket.txt
- ≡ 4.2_Macros en Racket.txt
- ① 4.3_Racket como Lenguaje orientado a lenguajes.md
-  GOTO-language.rkt

✓ 05_Enfoque educativo

- ≡ 5.1_Caracter educativo.txt
- ≡ 5.2_Ventajas.txt
- ≡ 5.3_Desventajas.txt

Filosofia detras

Racket es un lenguaje funcional concebido principalmente para:

Crear y experimentar con nuevos lenguajes

Propósitos educativos



Lenguajes funcionales

Un lenguaje de programación funcional es un paradigma de programación que se centra en el "qué hacer" en lugar de "cómo hacerlo".

Características principales:

Nivel de abstracion

Funciones como ciudadanos de primera clase

Inmutabilidad

Funciones puras

Recursión en lugar de bucles

Composición de funciones

1Evaluación perezosa contra evaluación estricta

Lenguajes funcionales

Nivel de abstracción:

```
(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)
```

```
#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;
}
```

Lenguajes funcionales

Funciones como ciudadanos de primera clase:

```
; 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
```

Lenguajes funcionales

Inmutabilidad:

```
; 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)
```

Lenguajes funcionales

Funciones puras:

Ejemplo de funciones pura

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

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

Ejemplo de funciones impura

```
(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
```


Lenguajes funcionales

Recursión en lugar de bucles:

```
; 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
```

≡ 1.1_Filosofia
detras.txt

≡ 1.2_Lenguajes
funcionale.txt

≡ 1.3_Historia de
Racket.txt

≡ 1.4_Soporte para
DSL.txt

Lenguajes funcionales

Composición de funciones:

```
; 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)
```

Lenguajes funcionales

Evaluación perezosa contra evaluación estricta:

Evaluacion perezosa:

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.

Evaluacion estrict:

Cada expresión se calcula inmediatamente cuando se encuentra en el código. Incluso los valores que podrían no ser utilizados nunca se calculan.

≡ 1.1_Filosofia
detras.txt

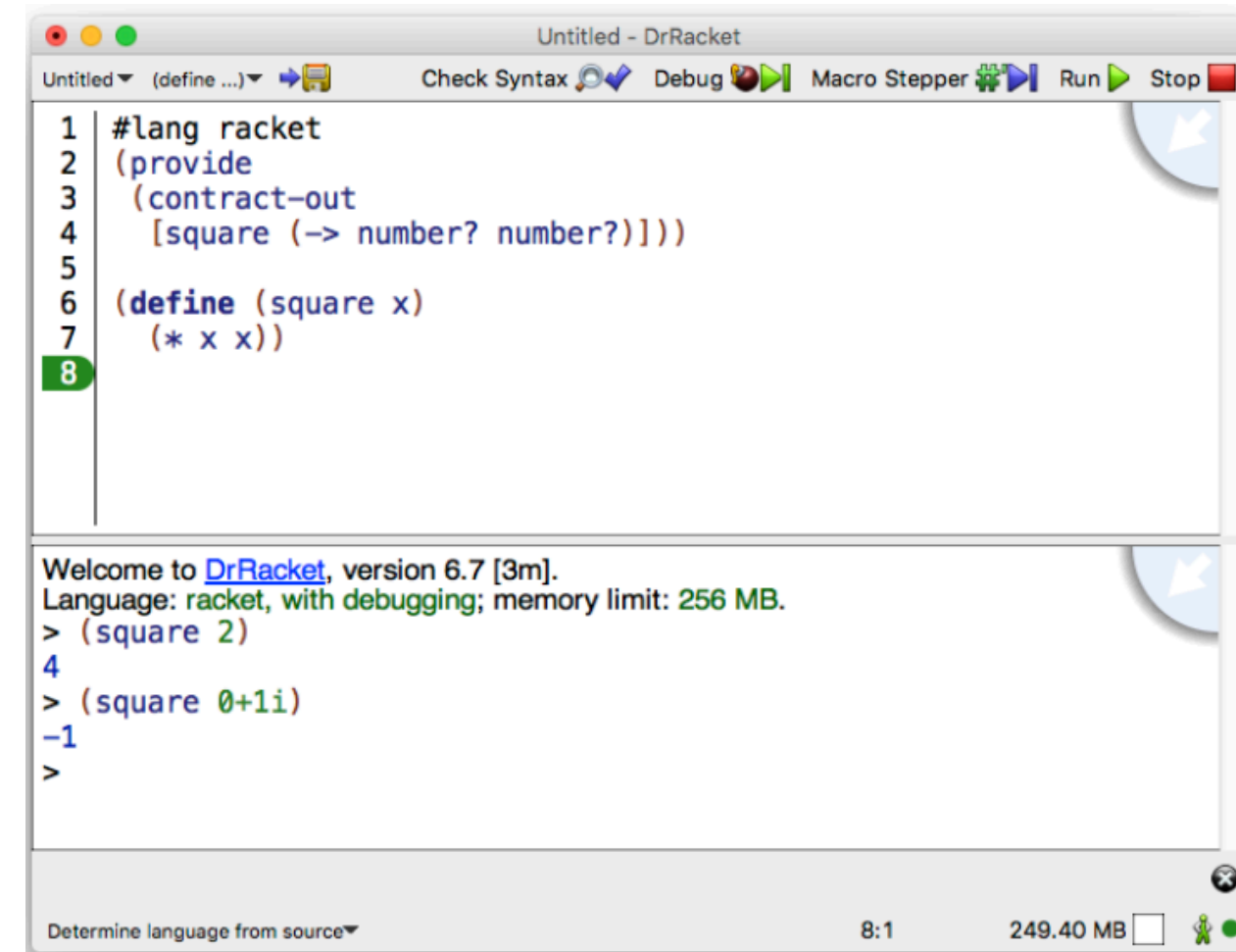
≡ 1.2_Lenguajes
funcionale.txt

≡ 1.3_Historia de
Racket.txt

≡ 1.4_Soporte para
DSL.txt

Historia de Racket

- Orígenes: Scheme y Lisp
- Desarrollo de Racket
- La evolución en un ecosistema completo
 - DrRacket
 - Bibliotecas potentes
 - Sporte para DSL
 - Herramientas para la investigacion
- Racket hoy
 - Educacion
 - Investigacion
 - Aplicaciones practicas



The screenshot shows the DrRacket IDE interface. The top window, titled 'Untitled - DrRacket', contains a Racket program with the following code:

```
1 #lang racket
2 (provide
3   (contract-out
4     [square (-> number? number?)]))
5
6 (define (square x)
7   (* x x))
8
```

The bottom window shows the output of the program:

```
Welcome to DrRacket, version 6.7 [3m].
Language: racket, with debugging; memory limit: 256 MB.
> (square 2)
4
> (square 0+1i)
-1
>
```

The status bar at the bottom indicates the language is 'Determine language from source', the zoom level is '8:1', and the memory usage is '249.40 MB'.

Soporte para DSL

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.



1.1_Filosofia
detras.txt



1.2_Lenguajes
funcionale.txt



1.3_Historia de
Racket.txt



1.4_Soporte para
DSL.txt

¿Qué son?

Las expresiones simbólicas son la forma de expresar estructuras de datos y código en Racket.

Son listas de símbolos y operadores con notación **prefija**.
Puede ser una cadena, un número, una función o una lista entre otros.

Características

Uniformidad

Simplicidad

Recursividad

Universalidad

✓ 02_Sintaxis basada en S-expresiones

≡ 2.1_Definición y características.txt

≡ 2.2_Cómo se utilizan.txt

≡ 2.2_Ejemplos básicos ✕

...

números

cadena

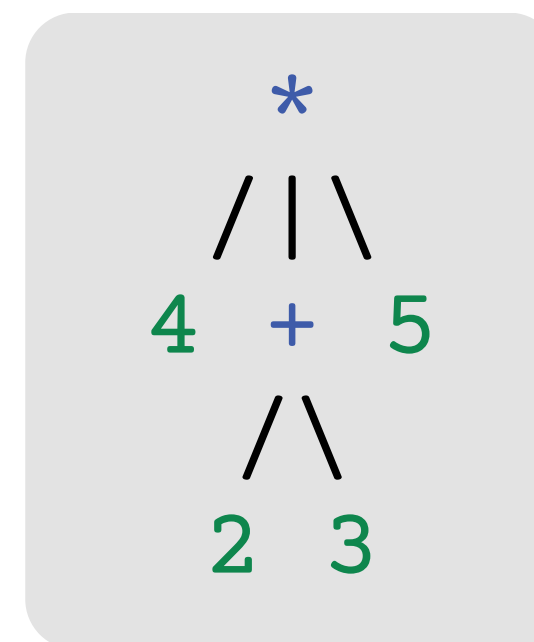
función

```
> -25
> 3.14
> "Hola mundo"
> (min 0.01 -5 20 1/2)
-5
> (* 4 (+ 2 3) 5)
100
```

operador

argumentos*

Evaluación en orden descendente



Asignación

```
(define identificador valor)
```

```
>(define x 5)  
>(define y (+ x 2))
```

Funciones

```
(define (nombre par1 ...) cuerpo)
```

```
>(define (suma' x)  
          (+ x square(y)))  
  
>suma' x y  
54
```


(lambda parámetros cuerpo)

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

```
>(define sumador n  
  (lambda (x) (+ x n)))
```

```
>(crear-sumador 2)  
#<procedure>
```

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

(lambda parámetros cuerpo)

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

```
>(define sumador n  
  (lambda (x) (+ x n)))
```

```
>(crear-sumador 2)  
#<procedure>
```

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

Clausura



Let

```
(let ([id valor]...) cuerpo)
```

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

Let*

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

LetRec

[illegible]

El tipo de una variable está asociado al su valor en ese momento.

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

Las funciones se pueden ejecutar con tipos de datos distintos.

```
> (cons true (cons "Str" 4))  
(#t "Str" 4)  
  
> (+ 1 "Hola")  
Error de tipo
```

Contratos

Permiten definir restricciones explícitas sobre los valores que una función puede aceptar o devolver que se verifican en tiempo de ejecución.

```
>(define/contract (maybe-invert x b
  (-> integer? boolean? integer?)
  (if b (-x) x))

>(maybe-invert 1 #t)
-1
>(maybe-invert #t 1)
Contract violation
expected: integer?
given: #f
```

Los tipos de las variables y expresiones se determinan y verifican durante la fase de **compilación**.

TypedRacket

Algunos tipos: *Integer*, *Real*, *String*, *Boolean*, *Listof*, *Pair* y *Any*.

```
( : 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 genérico  
(define (id x) x)
```

```
(define-type Tree (U leaf node))  
(struct leaf ([val : Number] )  
(struct node ([left : Tree] [right : Tree])
```

≡ 3.1_Tipo Dinámico.txt

≡ 3.2_Tipo Estático.txt

① 3.3_Comparación.m

🚦 3.4_Tipado Gradual.r

```
#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)
```

03_Sistemas de Tipos

3.1_Tipo Dinámico.txt

3.2_Tipo Estático.txt

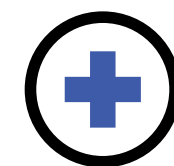
3.3_Comparación.m

3.4_Tipado Gradual.r

3.3_Comparación.txt

Características	Tipado Dinámico		Tipado Estático
	General	Contratos	
Verificación de Tipos	En ejecución	En ejecución	En compilación
Flexibilidad	Alta	Alta	Media
Seguridad	Baja	Media	Alta
Expresividad	Para escenarios simples y rápidos	Para validar situaciones complejas	Moderada, formalización detallada
Rendimiento	Bajo	Bajo	Alto
Uso en Racket	Predeterminado	<i>#lang typed/racket</i>	<i>require/contract</i>

Sistema Dinámico



Sistema Estático



Interoperabilidad

Adaptación Progresiva

Sobrecarga

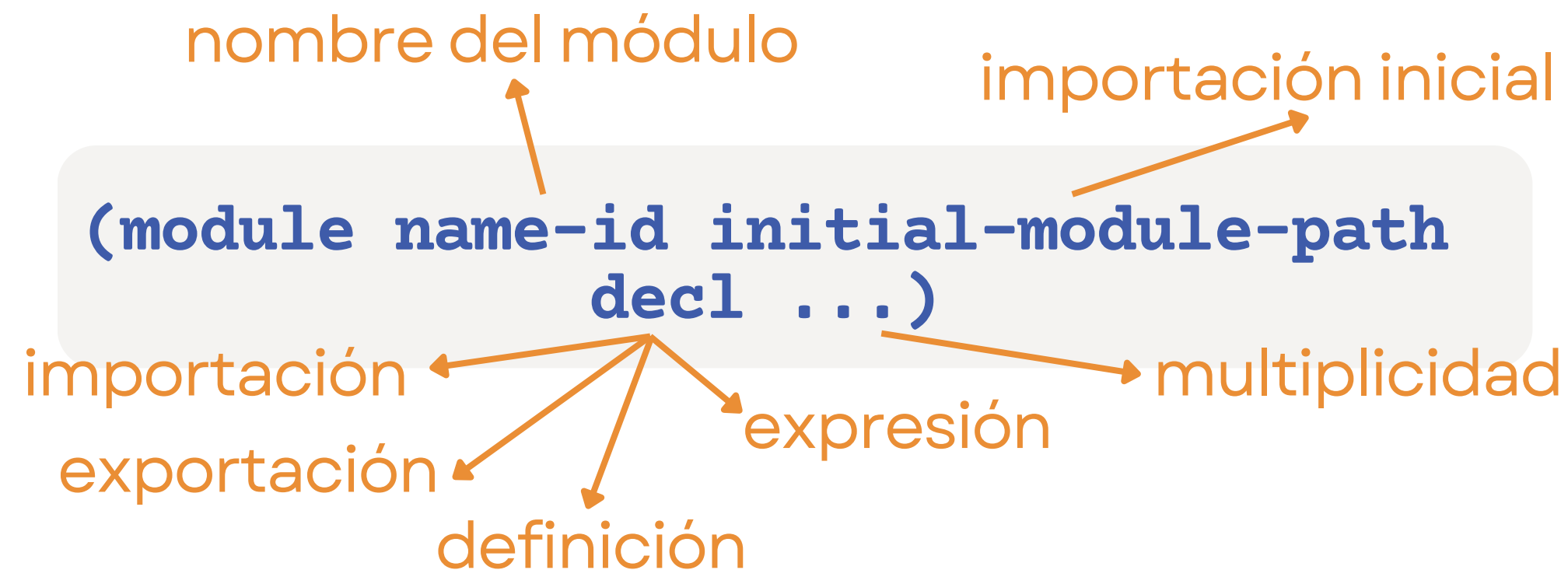
Ejemplo: Funcion con tipo estático espera algo de tipo T y recibe v de tipo dinámico. En tiempo de ejecucion se compruébale si T es de tipo v.

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

¿Qué son y cómo funcionan?

Fundamentales para la **organización** y el manejo de **dependencias**
Dividir el código en componentes **independientes** y **reutilizables**.

Sintáxis



Export = `provide`

Import = `require`

Ejemplo “cake.rkt” y “random-cake.rkt”

definimos el módulo
y exportamos

importamos y
usamos

cake.rkt (define ...)

```
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
```



random-cake.rkt (define ...)

```
1 #lang racket
2
3 (require "cake.rkt")
4
5 (print-cake (random 30))
6
```

Welcome to [DrRacket](#), version 8.14 [cs].
Language: racket, with debugging; memory limit: 128 MB.

```
      .....
    .-|||||||-.
    |           |
    -----
>
```

04_Módulos, Macros y su uso para la creación de DSL

≡ 4.1_Módulos.txt

≡ 4.2_Macros.txt

④ 4.3_Racket,
Lenguaje orientado
a lenguajes.md

🔗 GOTO-language.rkt

≡ 4.2_Macros.txt



¿Qué son y cómo funcionan?

Forma sintáctica con un **transformador** asociado que expande la forma **original** en formas **existentes**.

Higiénicas

Evita **errores** de colisión

transformamos
en varios
“display”

Ejemplo:

Permitir print de más de un argumento

macro-print.rkt (define ...)

```
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))))))
```

parseamos e
identificamos

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

04_Módulos, Macros y su uso para la creación de DSL

≡ 4.1_Módulos.txt

≡ 4.2_Macros.txt

④ 4.3_Racket,
Lenguaje orientado
a lenguajes.md

🌐 GOTO-language.rkt

④ 4.3_Racket como Lenguaje orientado a lenguajes.md ✕



¿Qué son los Lenguajes Específicos del dominio (DSL)?

Lenguaje de **alto** nivel de **abstracción** optimizado para resolver problemas **especializados**.

Racket como Meta-Lenguaje para DSLs

Racket destaca por su capacidad para crear otros lenguajes de programación por su sistema de **macros avanzado** y su **diseño modular**.

Setup Básico

Reader: sintáxis

Expander: semántica

04_Módulos, Macros y su uso para la creación de DSL

≡ 4.1_Módulos.txt

≡ 4.2_Macros.txt

④ 4.3_Racket,
Lenguaje orientado
a lenguajes.md

🔗 GOTO-language.rkt

🔗 GOTO-language.rkt



EXPANDER:

Definimos la sintaxis con
el nombre goto-language

El tipo de la variable

Para traducir múltiples
líneas de código

```
(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))
      Traducimos a lenguaje Racket #,@(map (lambda (line-number next-line-number command)
                                                (define name (make-line-name #'goto-language line-number))
                                                (define call-next-line
                                                  (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 ...)))))))
```

Funciones lambda para
crear el set de variables

Función para que ejecute todas las líneas
sucesivamente

04_Módulos, Macros y su uso para la creación de DSL

≡ 4.1_Módulos.txt

≡ 4.2_Macros.txt

④ 4.3_Racket,
Lenguaje orientado
a lenguajes.md

🔗 GOTO-language.rkt

🔗 GOTO-language.rkt



READER:

```
#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))))))
```

exportamos

manejo de errores

convierte los datos en objeto de sintaxis

importa las transformaciones

04_Módulos, Macros y su uso para la creación de DSL

≡ 4.1_Módulos.txt

≡ 4.2_Macros.txt

④ 4.3_Racket,
Lenguaje orientado
a lenguajes.md

🔗 GOTO-language.rkt

🔗 GOTO-language.rkt



DEMO:

$f(x) = \max \{ n : n^2 \leq x \}$

[A] $Z \leftarrow Y^2$

IF $Z \leq X$ GOTO C ELSE GOTO B

[B] $Y \leftarrow Y - 1$

PRINT Y

RETURN

[C] $Y \leftarrow Y + 1$

GOTO A

usamos el lenguaje GOTO

```
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

>

Principales enfoques de Racket en la enseñanza

- ≡ 5.1_Enfoque educativo.txt
- ≡ 5.2_Ventajas.txt
- ≡ 5.3_Desventajas.txt

Fundamentos de la programación

Sintaxis minimalista

Evaluación de expresiones

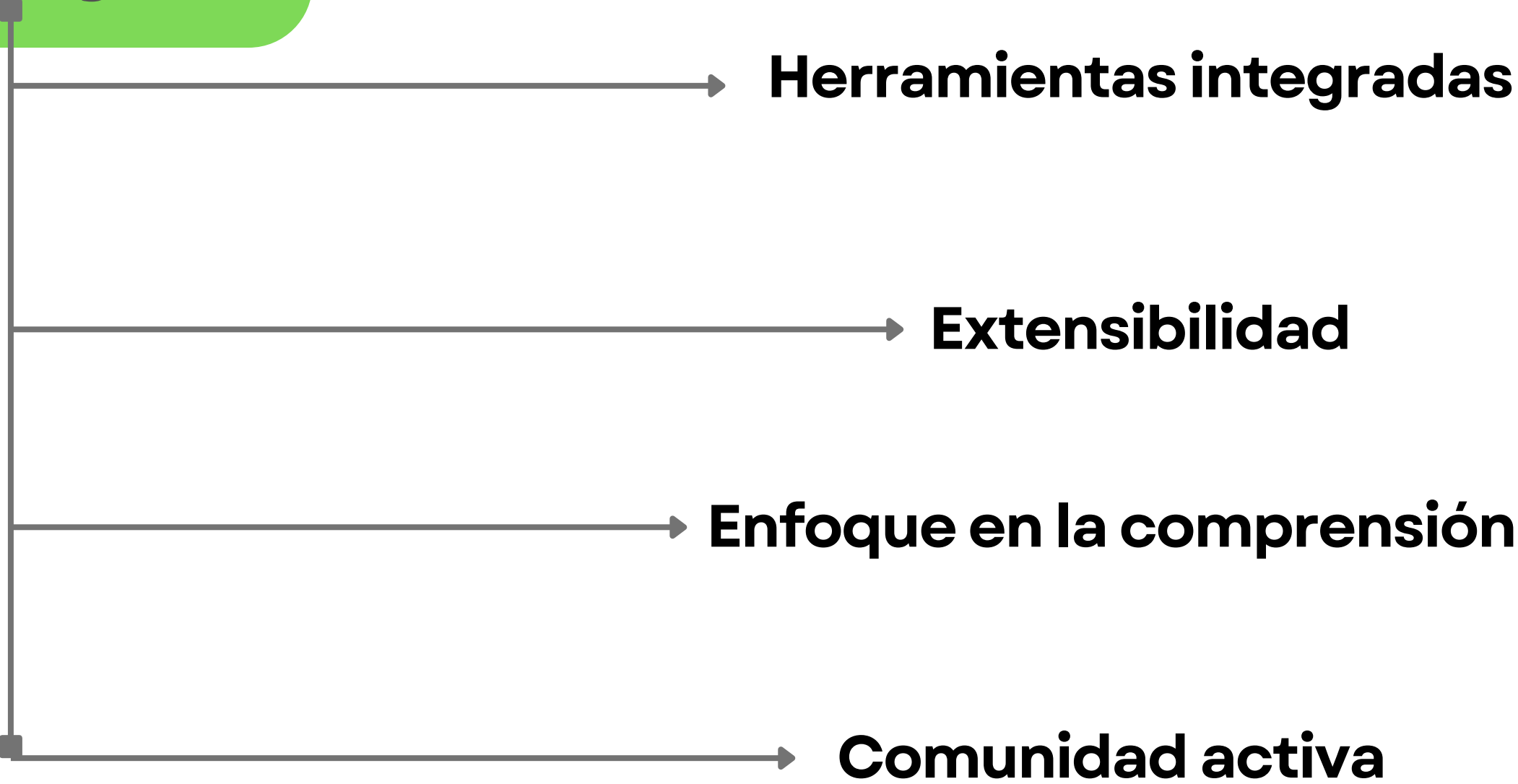
Funciones como “ciudadanos de primera clase”

Resolución de problemas

Modelado de conceptos abstractos

Desarrollo de algoritmos

Ventajas



Desventajas

Popularidad

Rendimiento

Integración