# Lambda

Un sistema computacional basado en el cálculo $\lambda$

por

Eduardo Acuña Yeomans

Universidad de Sonora
Licenciatura en Ciencias de la Computación
Departamento de Matemáticas
División de Ciencias Exactas y Naturales

*Hermosillo, Sonora*
2016

# Índice general

# Capítulo 1

# Introducción

El cálculo $\lambda$ es estudiado usualmente desde un punto de vista matemático, ya sea formulado como una teoría ecuacional o como un sistema de reducción, esto implica que antes de poder trabajar con él, es necesario entender su formalización.

Este documento presenta una alternativa a esta manera de trabajar con el cálculo $\lambda$ ya que describe la implementación de un sistema computacional en lugar de uno matemático. El sistema computacional `Lambda`, al ser un programa de computadora, puede ser utilizado para estudiar y trabajar con el cálculo $\lambda$ de manera interactiva.

## 1.1 Programas literarios

Este documento no solo describe la implementación de `Lambda`, también *es* la implementación. La herramienta `noweb` para "programación literaria" genera tanto el documento como el código para `Lambda` a partir de un mismo archivo fuente. Este archivo consiste de pedazos intercalados de prosa y códigos etiquetados llamados *fragmentos*. Los fragmentos son escritos en un orden conveniente para describir el programa, es decir, el orden en el que aparecen en este documento, en lugar de un orden determinado por el lenguaje de programación de la implementación. El programa `noweave` toma el archivo fuente y produce un archivo en LaTeX, el cuál es compilado para producir este documento. El programa `notangle` extrae todo el código del archivo fuente y lo vacía en varios archivos en un orden adecuado para su compilación o interpretación.

Los fragmentos contienen código fuente o referencias a otros fragmentos. Los fragmentos de definiciones son precedidos por sus etiquetas entre paréntesis angulares. Por ejemplo, el código

1a ⟨*suma elementos del vector* v 1a⟩≡ 2▸
```
(define total
  (let itera ((suma 0)
              (i    0))
    (if (= i (vector-length v))
        suma
        (itera ⟨aumenta la suma total 1b⟩ (+ i 1)))))
```

Define:
`total`, utilizado en fragmento 2.

1b ⟨*aumenta la* suma *total* 1b⟩≡ (1a)
```
(+ suma (vector-ref v i))
```

suma los elementos del vector v. La utilización de fragmentos es mostrada como ⟨*aumenta la* suma *total* 1b⟩ del fragmento 1a. Varios fragmentos pueden tener la misma etiqueta; notangle concatena sus contenidos para producir un único fragmento. noweave identifica esta concatenación utilizando el símbolo +≡ en lugar de ≡ en fragmentos continuados:

2 ⟨*suma elementos del vector* v 1a⟩+≡ ◁1a

```
(printf "La suma de los elementos de ~a es ~a~%" v total)
```

Utiliza total 1a.

El programa notangle obtiene el código de un programa expandiendo un fragmento. Si en el contenido de este fragmento hay referencias a otros fragmentos, estos también son expandidos y así sucesivamente.

Los fragmentos incluyen descripciones de definición y referencia de identificadores en la parte inferior del código e indicadores de inclusión y continuación de fragmentos a la derecha de su etiqueta para auxiliar al lector en la navegación del documento, los números en las descripciones e indicadores se refieren a páginas del documento.

## 1.2 Cómo leer este documento

Lambda está escrito para leerse de inicio a fin, sin embargo algunas partes del programa son tediosas de leer y otras, a pesar de poderse considerar interesantes, no se relacionan propiamente con el cálculo $\lambda$, si no con aspectos relacionados con programación.

Ya que los que estamos familiarizadas con la programación acostumbramos leer una implementación en el orden impuesto por el lenguaje, es posible que cueste trabajo en un inicio entender el código presente en este documento, sin embargo, una manera de leer el código es identificando los archivos que son generados por notangle, estos tienen etiquetas con extensión .rkt y su definición establece los fragmentos de código que son parte del archivo.

Otra manera alternativa de leer este documento es hojearlo por completo prestando atención solo a la primer parte de cada capítulo para tener una idea global del programa. Posteriormente leer con más cuidado desde el inicio. Cualquiera que sea el método de lectura puede ser importante correr Lambda y explorarlo desde el punto de vista de un usuario.

## 1.3 Estructura del programa

La implementación de Lambda se divide en tres componentes:
**lambda** Una biblioteca con las estructuras y funciones asociadas al cálculo $\lambda$;
**lambda-gui** Una interfaz gráfica que permite estudiar el cálculo de manera interactiva;
**lambda-init** Una serie de procedimientos modificables por el usuario.

La componente referida como **lambda** contiene la implementación de los aspectos fundamentales del cálculo, por ejemplo, la definición de sus expresiones y procedimientos para leerlas, escribirlas y manipularlas.

La componente referida como **lambda-gui** contiene las estructuras y procedimientos para manejar las ventanas de la interfaz gráfica, la cuál hace uso de **lambda** para funcionar.

La componente referida como **lambda-init** contiene extensiones a **lambda** las cuales pueden ser modificadas y aumentadas por el usuario sin miedo a que el sistema deje de funcionar.

# Capítulo 2

# Piezas fundamentales

## 2.1 Objetos elementales

Existen diversas clases de objetos que pueden existir en el mundo del cálculo $\lambda$, estas clases de objetos representan ideas abstractas asociadas a cadenas de un lenguaje formal denotado $\Lambda$. Este lenguaje formal determina qué secuencias de símbolos son válidas expresar dentro del cálculo.

**Definición de expresión** Las clases de objetos cuya representación simbólica son parte de $\Lambda$ son llamados *términos*, en este programa se implementan los tres tipos de términos más básicos del cálculo $\lambda$: las *variables*, las *abstracciones* y las *aplicaciones*. Las clases de objetos del sistema `Lambda` tienen como representación simbólica elementos del lenguaje formal $\Omega$ llamados *expresiones*, los elementos de $\Omega$ incluyen a todos los términos y a representaciones de otros dos tipos de objetos: los *huecos* y las *metainstrucciones*. El predicado `expresión?` determina si un objeto tiene representación simbólica en $\Omega$.

3a ⟨*Definición de expresión* 3a⟩≡ (11)
```
(define/contract (expresión? x)
  (any/c . -> . boolean?)
  (or (variable? x) (abstracción? x) (aplicación? x)
      (hueco? x) (metainstrucción? x)))
```
Define:
  `expresión?`, utilizado en fragmentos 4–6, 8a, 13, 14, 17b, 19a, 20d, 23b, 27, 28, y 31a.
Utiliza `abstracción` 4c, `aplicación` 5a, `hueco` 5c, `metainstrucción` 6a, y `variable` 3b.

**Definición de variable** Las variables son una clase de objetos sencilla, se conforman de un *nombre* el cuál es usado para referirnos a ellas, por ejemplo, `Eduardo` es un nombre que puede referirse a una variable y usualmente diremos que `Eduardo` *es* la variable. Las variables son implementadas como una estructura compuesta de una cadena de caracteres. Utilizo el comando `define-struct/contract` para definir una estructura cuyo constructor verifica que el valor del campo `nombre` satisface el predicado `string?`.

3b ⟨*Definición de variable* 3b⟩≡ (11)
```
(define-struct/contract variable
  ([nombre string?])
  #:transparent
  #:mutable)
```

Define:
variable, utilizado en fragmentos 3–10, 12, 15–17, 19–21, 23, 26–28, 32a, 37, 47, y 57–60.

Al programar las variables de esta manera Racket define varios procedimientos: variable es un constructor que recibe una cadena de caracteres y regresa un objeto de este tipo; variable? es un predicado que recibe un valor y determina si es un valor de tipo variable; variable-nombre es un selector que dado un valor de tipo variable regresa la cadena de caracteres del campo nombre. La palabra clave #:transparent permite imprimir un valor de tipo variable de manera legible en el entorno interactivo de Racket, mientras que la palabra clave #:mutable hace que los campos muedan ser modificados, en este caso con el procedimiento set-variable-nombre!.

Para verificar que esta implementación tiene la funcionalidad esperada se define el caso de prueba "Implementación de variables".

4a ⟨*Casos de prueba* 4a⟩≡ (33) 4d▸
```
(test-case "Implementación de variables"
  (check-true (variable? (variable "x")))
  (check-false (variable? 42))
  (check-exn exn:fail:contract? (lambda () (variable 42)))
  (check-equal? (variable-nombre (variable "x")) "x"))
```
Utiliza variable 3b.

La definición de los casos de prueba se realiza utilizando los procedimientos de la biblioteca rackunit, la cual es invocada con el comando require.

4b ⟨*Dependencias de casos de prueba* 4b⟩≡ (33) 7b▸
```
(require rackunit)
```

**Definición de abstracción** Las abstracciones son clases de objetos compuestos que representan el concepto de *función*, se conforman de un *argumento* y un *cuerpo*, el argumento es una variable y el cuerpo es una expresión. El constructor de abstracciones verifica que las partes de la abstracción son del tipo adecuado por medio de los predicados variable? y expresión?.

4c ⟨*Definición de abstracción* 4c⟩≡ (11)
```
(define-struct/contract abstracción
  ([argumento variable?] [cuerpo expresión?])
  #:transparent
  #:mutable)
```
Define:
abstracción, utilizado en fragmentos 3–5, 7, 9, 12, 17–19, 21, 23, 26–28, 31a, 32a, 37, 47, y 57–60.
Utiliza expresión? 3a y variable 3b.

Se verifica el correcto funcionamiento de la implementación de las abstracciones con el caso de prueba "Implementación de abstracciones".

4d ⟨*Casos de prueba* 4a⟩+≡ (33) ◂4a 5b▸
```
(test-case "Implementación de abstracciones"
  (check-true (abstracción? (abstracción (variable "x") (variable "y"))))
  (check-false (abstracción? (variable "x")))
  (check-exn exn:fail:contract? (lambda () (abstracción "x" "y")))
  (check-equal? (abstracción-argumento (abstracción (variable "x") (variable "y")))
                (variable "x"))
  (check-equal? (abstracción-cuerpo (abstracción (variable "x") (variable "y")))
```

```
                    (variable "y")))
```
Utiliza abstracción 4c y variable 3b.

**Definición de aplicación** Al igual que las abstracciones, las aplicaciones también son clases de objetos compuestos, sin embargo estos se conforman de dos expresiones cualesquiera llamadas *operador* y *operando*. El constructor de aplicaciones verifica que las partes de la aplicación son del tipo adecuado utilizando el predicado expresión?.

5a ⟨*Definición de aplicación* 5a⟩≡ (11)
```
(define-struct/contract aplicación
  ([operador expresión?] [operando expresión?])
  #:transparent
  #:mutable)
```
Define:
aplicación, utilizado en fragmentos 3a, 5–7, 9c, 10a, 12, 17, 18, 20, 21, 23, 26–28, 31a, 32a, 37, 47, y 57–60.
Utiliza expresión? 3a.

Se verifica que la implementación es correcta con el caso de prueba "Implementación de aplicaciones".

5b ⟨*Casos de prueba* 4a⟩+≡ (33) ◂4d 5d▸
```
(test-case "Implementación de aplicaciones"
  (check-true (aplicación? (aplicación (variable "x") (variable "y"))))
  (check-false (aplicación? (abstracción (variable "x") (variable "y"))))
  (check-exn exn:fail:contract? (lambda () (aplicación "x" "y")))
  (check-equal? (aplicación-operador (aplicación (variable "x") (variable "y")))
                (variable "x"))
  (check-equal? (aplicación-operando (aplicación (variable "x") (variable "y")))
                (variable "y")))
```
Utiliza abstracción 4c, aplicación 5a, y variable 3b.

**Definición de hueco** Los huecos son una clase de objetos que representan información faltante. Las expresiones que contienen huecos son útiles únicamente para describir expresiones incompletas o expresiones cuya información faltante será descrita posteriormente. La implementación de los huecos consiste en una estructura sin campos.

5c ⟨*Definición de hueco* 5c⟩≡ (11)
```
(define-struct/contract hueco
  ()
  #:transparent)
```
Define:
hueco, utilizado en fragmentos 3a, 5d, 7d, 10c, 18, 21, 23, 26–28, 32a, 37, 47, y 59a.

La verificación de que esta implementación es correcta se realiza con el caso de prueba "Implementación de huecos" y es más sencilla que en las anteriores clases de objetos.

5d ⟨*Casos de prueba* 4a⟩+≡ (33) ◂5b 6b▸
```
(test-case "Implementación de huecos"
  (check-true (hueco? (hueco)))
  (check-false (hueco? 42)))
```
Utiliza hueco 5c.

**Definición de metainstrucción** La última clase de objetos de las expresiones de Lambda es las metainstrucciones, estas representan operaciones que manipulan otras expresiones y pueden generar como resultado objetos que no cumplen con el predicado expresión?. La implementación de las metainstrucciones se divide en dos partes: primero la implementación de las expresiones que denotan una invocación de una metainstrucción y posteriormente la definición de una metainstrucción como procedimiento de Racket que se encarga de realizar un cómputo.

La invocación de metainstrucciones se conforma de un *nombre* y de una lista de expresiones llamada *argumentos*. El constructor de invocaciones de metainstrucciones verifica que sus campos son del tipo adecuado utilizando el predicado string? y (*list/c expresión?).

6a ⟨*Definición de metainstrucción* 6a⟩≡ (11) 6c▸

```
(define-struct/contract metainstrucción
  ([nombre string?] [argumentos (*list/c expresión?)])
  #:transparent
  #:mutable)
```

Define:
metainstrucción, utilizado en fragmentos 3a, 6, 7a, 10, 20, 21, 23, 26–28, 31a, 32a, 37, 47, 50, y 57–60.
Utiliza expresión? 3a.

El caso de prueba "Implementación de metainstrucciones" verifica que esta implementación funciona correctamente.

6b ⟨*Casos de prueba* 4a⟩+≡ (33) ◂5d 7a▸

```
(test-case "Implementación de metainstrucciones"
  (check-true (metainstrucción? (metainstrucción "f" (list (variable "x")))))
  (check-false (metainstrucción? (aplicación (variable "x") (variable "y"))))
  (check-exn exn:fail:contract?
             (lambda () (metainstrucción (variable "f") (variable "x"))))
  (check-equal? (metainstrucción-nombre
                 (metainstrucción "f" (list (variable "x")))) "f")
  (check-equal? (metainstrucción-argumentos
                 (metainstrucción "f" (list (variable "x"))))
                (list (variable "x"))))
```

Utiliza aplicación 5a, metainstrucciones 6c, metainstrucción 6a, y variable 3b.

La definición de metainstrucciones consiste en la definición de un procedimiento de Racket y la asociación del nombre de la metainstrucción a este procedimiento en la tabla hash llamada metainstrucciones. Esta funcionalidad se implementa con el macro define-metainstrucción y en la sección 4 se muestra la implementación de varias metainstrucciones.

6c ⟨*Definición de metainstrucción* 6a⟩+≡ (11) ◂6a

```
(define metainstrucciones (make-hash))

(define-syntax define-metainstrucción
  (syntax-rules ()
    [(define-metainstrucción (nombre . argumentos)
       cómputo cómputos ...)
     (begin
      (define (nombre . argumentos)
        cómputo cómputos ...)
      (hash-set! metainstrucciones
                 (symbol->string 'nombre)
                 nombre))]))
```

Define:
  define-metainstrucción, utilizado en fragmentos 7a, 32a, 50, y 57–60.
  metainstrucciones, utilizado en fragmentos 6b, 7a, 10c, 23b, 31a, 50, y 51.
Utiliza metainstrucción 6a.

Para verificar que el macro define-metainstrucción tiene la funcionalidad esperada se define el caso de prueba "Definición de metainstrucciones".

7a ⟨*Casos de prueba* 4a⟩+≡ (33) ◂6b 7d▸
```
(test-case "Definición de metainstrucciones"
  (check-equal? (syntax->datum (expand-only #'(define-metainstrucción (f x)
                                                (if (variable? x) 1 0))
                                              (list #'define-metainstrucción)))
                '(begin
                  (define (f x) (if (variable? x) 1 0))
                  (hash-set! metainstrucciones (symbol->string 'f) f))))
```
Utiliza define-metainstrucción 6c, metainstrucciones 6c, metainstrucción 6a, y variable 3b.

Para poder expander el macro en el caso de prueba es necesario hacer uso de la biblioteca macro-debugger/expand.

7b ⟨*Dependencias de casos de prueba* 4b⟩+≡ (33) ◂4b
```
(require macro-debugger/expand)
```

**Definición de término** Teniendo implementadas las estructuras con representación simbólica en Ω es conveniente definir un procedimiento para determinar si una expresión es un término. El predicado término? inspecciona una expresión para verificar que se conforma únicamente de variables, abstracciones o aplicaciones.

7c ⟨*Definición de término* 7c⟩≡ (11)
```
(define (término? x)
  (cond [(variable? x)    true]
        [(abstracción? x) (término? (abstracción-cuerpo x))]
        [(aplicación? x)  (and (término? (aplicación-operador x))
                               (término? (aplicación-operando x)))]
        [else             false]))
```
Define:
  término?, utilizado en fragmentos 7d, 8a, y 57a.
Utiliza abstracción 4c, aplicación 5a, y variable 3b.

El caso de prueba "Predicado para términos" se encarga de verificar que la funcionalidad de este procedimiento es correcta.

7d ⟨*Casos de prueba* 4a⟩+≡ (33) ◂7a 8c▸
```
(test-case "Predicado para términos"
  (check-true
   (término? (abstracción (variable "x")
                          (aplicación (variable "x") (variable "x")))))
  (check-false
   (término? (abstracción (variable "x") (aplicación (variable "x") (hueco))))))
```
Utiliza abstracción 4c, aplicación 5a, hueco 5c, término? 7c, y variable 3b.

**Comparación de expresiones** El uso del mecanismo `define-struct` de `Racket` tiene la ventaja de que la verificación de que dos expresiones son del mismo tipo y tienen las mismas subexpresiones se puede realizar con el procedimiento genérico `equal?` y al igual que todos los objetos de cualquier tipo, la comparación `eq?` puede ser utilizada para determinar si dos objetos de cualquier tipo son el mismo objeto en memoria. Sin embargo, es útil implementar otros procedimientos que comparen aspectos particulares de dos expresiones, por ejemplo, si dos metainstrucciones tienen la misma cantidad de argumentos, si dos abstracciones tienen el mismo cuerpo, etc.

Se definen los predicados `expresión=?` y `término=?` como procedimientos que además de utilizar `equal?` verifican el tipo de valores que reciben.

8a ⟨*Comparación de expresiones* 8a⟩≡ (11)

```
(define/contract (expresión=? e1 e2)
  (expresión? expresión? . -> . boolean?)
  (equal? e1 e2))

(define/contract (término=? t1 t2)
  (término? término? . -> . boolean?)
  (equal? t1 t2))
```

Define:
`expresión=?`, utilizado en fragmento 9.
`término=?`, nunca utilizado.
Utiliza `expresión?` 3a y `término?` 7c.

**Comparación de variables** Dos variables con el mismo nombre son consideradas diferentes ya que la importancia de la variable radica en cómo es utilizada. Es por ello que es conveniente referirnos a todas las variables con el mismo nombre como "átomo" (ya que son objetos sin estructura, es decir, no se puede separar en partes). Los predicados `variable=?` y `átomo=?` permiten comparar dos variables de acuerdo a estos criterios.

8b ⟨*Comparación de variables* 8b⟩≡ (11)

```
(define/contract (variable=? v1 v2)
  (variable? variable? . -> . boolean?)
  (eq? v1 v2))

(define/contract (átomo=? v1 v2)
  (variable? variable? . -> . boolean?)
  (equal? v1 v2))
```

Define:
`variable=?`, utilizado en fragmento 8c.
`átomo=?`, utilizado en fragmentos 8c y 9a.
Utiliza `variable` 3b.

Para verificar el correcto funcionamiento de estos predicados se define el caso de prueba "`Predicados de variables`".

8c ⟨*Casos de prueba* 4a⟩+≡ (33) ◂7d 9b▸

```
(test-case "Predicados de variables"
  (let ([v (variable "x")])
    (check-true (variable=? v v)))
  (check-false (variable=? (variable "x") (variable "x")))
  (check-true (átomo=? (variable "x") (variable "x")))
  (check-false (átomo=? (variable "x") (variable "y"))))
```

Utiliza `variable` 3b, `variable=?` 8b, y `átomo=?` 8b.

**Comparación de abstracciones** En el caso de las abstracciones consideramos dos comparaciones: entre argumentos y entre cuerpos.

9a ⟨*Comparación de abstracciones* 9a⟩≡ (11)
```
(define/contract (argumento=? a1 a2)
  (abstracción? abstracción? . -> . boolean?)
  (átomo=? (abstracción-argumento a1)
           (abstracción-argumento a2)))

(define/contract (cuerpo=? a1 a2)
  (abstracción? abstracción? . -> . boolean?)
  (expresión=? (abstracción-cuerpo a1)
               (abstracción-cuerpo a2)))
```
Define:
argumento=?, utilizado en fragmento 9b.
cuerpo=?, utilizado en fragmento 9b.
Utiliza abstracción 4c, expresión=? 8a, y átomo=? 8b.

Para verificar el correcto funcionamiento de estos predicados se define el caso de prueba "Predicados de abstracciones".

9b ⟨*Casos de prueba* 4a⟩+≡ (33) ◂8c 10a▸
```
(test-case "Predicados de abstracciones"
  (check-true (argumento=? (abstracción (variable "x") (variable "y"))
                           (abstracción (variable "x") (variable "z"))))
  (check-false (argumento=? (abstracción (variable "y") (variable "x"))
                            (abstracción (variable "z") (variable "x"))))
  (check-true (cuerpo=? (abstracción (variable "y") (variable "x"))
                        (abstracción (variable "z") (variable "x"))))
  (check-false (cuerpo=? (abstracción (variable "x") (variable "y"))
                         (abstracción (variable "x") (variable "z")))))
```
Utiliza abstracción 4c, argumento=? 9a, cuerpo=? 9a, y variable 3b.

**Comparación de aplicaciones** De manera similar a las abstracciones, se implementan predicados para verificar si dos aplicaciones comparten operador u operando.

9c ⟨*Comparación de aplicaciones* 9c⟩≡ (11)
```
(define/contract (operador=? a1 a2)
  (aplicación? aplicación? . -> . boolean?)
  (expresión=? (aplicación-operador a1)
               (aplicación-operador a2)))

(define/contract (operando=? a1 a2)
  (aplicación? aplicación? . -> . boolean?)
  (expresión=? (aplicación-operando a1)
               (aplicación-operando a2)))
```
Define:
operador=?, utilizado en fragmento 10a.
operando=?, utilizado en fragmento 10a.
Utiliza aplicación 5a y expresión=? 8a.

El caso de prueba "Predicados de aplicaciones" verifica que estos dos predicados sean correctos.

10a ⟨*Casos de prueba* 4a⟩+≡ (33) ◂9b 10c▸
```
(test-case "Predicados de aplicaciones"
  (check-true (operador=? (aplicación (variable "x") (variable "y"))
                          (aplicación (variable "x") (variable "z"))))
  (check-false (operador=? (aplicación (variable "y") (variable "x"))
                           (aplicación (variable "z") (variable "x"))))
  (check-true (operando=? (aplicación (variable "y") (variable "x"))
                          (aplicación (variable "z") (variable "x"))))
  (check-false (operando=? (aplicación (variable "x") (variable "y"))
                           (aplicación (variable "x") (variable "z")))))
```
Utiliza aplicación 5a, operador=? 9c, operando=? 9c, y variable 3b.

**Comparación de metainstrucciones** Para las metainstrucciones se implementan tres predicados de comparación: comparación de nombres, comparación de cantidad de argumentos y comparación de argumentos.

10b ⟨*Comparación de metainstrucciones* 10b⟩≡ (11)
```
(define/contract (nombre=? m1 m2)
  (metainstrucción? metainstrucción? . -> . boolean?)
  (string=? (metainstrucción-nombre m1)
            (metainstrucción-nombre m2)))

(define/contract (cantidad-argumentos=? m1 m2)
  (metainstrucción? metainstrucción? . -> . boolean?)
  (= (length (metainstrucción-argumentos m1))
     (length (metainstrucción-argumentos m2))))

(define/contract (argumentos=? m1 m2)
  (metainstrucción? metainstrucción? . -> . boolean?)
  (equal? (metainstrucción-argumentos m1)
          (metainstrucción-argumentos m2)))
```
Define:
argumentos=?, utilizado en fragmento 10c.
cantidad-argumentos=?, utilizado en fragmento 10c.
nombre=?, utilizado en fragmento 10c.
Utiliza metainstrucción 6a.

El caso de prueba "Predicados de metainstrucciones" verifica que la implementación es correcta.

10c ⟨*Casos de prueba* 4a⟩+≡ (33) ◂10a 15b▸
```
(test-case "Predicados de metainstrucciones"
  (check-true (nombre=? (metainstrucción "f" null)
                        (metainstrucción "f" (list (variable "x")))))
  (check-false (nombre=? (metainstrucción "f" (list (variable "x")))
                         (metainstrucción "g" null)))
  (check-true (cantidad-argumentos=? (metainstrucción "f" (list (variable "x")))
                                     (metainstrucción "g" (list (hueco)))))
  (check-false (cantidad-argumentos=? (metainstrucción "f" (list (variable "x")))
                                      (metainstrucción "f" null)))
  (check-true (argumentos=? (metainstrucción "f" (list (variable "x")
                                                       (variable "y")))
```

```
                              (metainstrucción "g" (list (variable "x")
                                                         (variable "y")))))
    (check-false (argumentos=? (metainstrucción "f" (list (variable "x")))
                               (metainstrucción "f" (list (variable "x")
                                                          (variable "y"))))))
```

Utiliza `argumentos=?` 10b, `cantidad-argumentos=?` 10b, `hueco` 5c, `metainstrucciones` 6c, `metainstrucción` 6a, `nombre=?` 10b, y `variable` 3b.

**Estructuras de las expresiones** Los fragmentos de código presentados anteriormente conforman la implementación de las clases de objetos con representación simbólica en Ω, es decir, la implementación de las expresiones del sistema `Lambda`. El orden de los fragmentos de código es de la siguiente manera:

11 ⟨*Estructuras de las expresiones* 11⟩≡ (32b)
- ⟨*Definición de expresión* 3a⟩
- ⟨*Definición de término* 7c⟩
- ⟨*Comparación de expresiones* 8a⟩
- ⟨*Definición de variable* 3b⟩
- ⟨*Comparación de variables* 8b⟩
- ⟨*Definición de abstracción* 4c⟩
- ⟨*Comparación de abstracciones* 9a⟩
- ⟨*Definición de aplicación* 5a⟩
- ⟨*Comparación de aplicaciones* 9c⟩
- ⟨*Definición de hueco* 5c⟩
- ⟨*Definición de metainstrucción* 6a⟩
- ⟨*Comparación de metainstrucciones* 10b⟩

## 2.2 Lenguaje de expresiones

**De Ω a expresiones y viceversa** Tener implementadas las estructuras de las expresiones es lo único necesario para poder escribir algoritmos que analicen y manipulen expresiones en el sistema `Lambda`, sin embargo es inconveniente definir expresiones en particular y leer expresiones que sean resultado de algún algoritmo.

Como se menciona al inicio de esta sección las estructuras implementadas son representaciones computacionales de elementos del lenguaje formal Ω, estos elementos son cadenas de símbolos los cuáles son más amenos de escribir y leer que utilizando los constructores y representaciones textuales de las estructuras programadas en `Racket`.

La representación simbólica de una variable con nombre "`x`" es $x$, en el caso de una abstracción de la forma `(abstracción x M)` su representación simbólica es la cadena $(\lambda x.M)$ donde $x$ y $M$ son las representaciones simbólicas de `x` y `M`, de manera similar, una aplicación de la forma `(aplicación M N)` tiene como representación simbólica $(M\,N)$. Estas notaciones son estándar en el estudio formal del cálculo $\lambda$.

En el caso de los huecos y las metainstrucciones, tomaremos una notación simbólica inventada, para un hueco será `[ ]` y para una metainstrucción de la forma `(metainstrucción f (list M1 ... Mn))` será $f[M_1, ..., M_n]$.

Se define el lenguaje libre de contexto Ω de manera informal de la siguiente manera. Sea $D = \{$'(', ')', '[', ']', '.', ',', 'λ', ' '$\}$ un conjunto de símbolos delimitadores donde ' ' es el símbolo de espacio en blanco y sea $N$ el conjunto de nombres compuestos de uno o más

símbolos $a \notin D$

$$\begin{aligned} x &\in \Omega && \text{Si } x \in N \\ (\lambda x.P) &\in \Omega && \text{Si } x \in N \text{ y } P \in \Omega \\ (P\,Q) &\in \Omega && \text{Si } P,\ Q \in \Omega \\ [\quad] &\in \Omega \\ x[P_1, P_2, ..., P_n] &\in \Omega && \text{Si } x \in N \text{ y } P_i \in \Omega \end{aligned}$$

Conociendo la sintaxis de los elementos de $\Omega$ podemos comparar representaciones diferentes de una misma expresión, por ejemplo, un término muy famoso es el *combinador Y*, simbólicamente es escrito:

$$(\lambda f.((\lambda x.(f\,(x\,x)))(\lambda x.(f\,(x\,x)))))$$

mientras que su representación con las estructuras que se implementaron es:

12 ⟨*El combinador Y* 12⟩≡

```
(abstracción (variable "f")
             (aplicación (abstracción (variable "x")
                                      (aplicación (variable "f")
                                                  (aplicación (variable "x")
                                                              (variable "x"))))
                         (abstracción (variable "x")
                                      (aplicación (variable "f")
                                                  (aplicación (variable "x")
                                                              (variable "x"))))))
```

Utiliza abstracción 4c, aplicación 5a, y variable 3b.

También se contemplan abusos de notación a la representación simbólica, las cuales son:

$$(\lambda x_1.(\lambda x_2.(...(\lambda x_n.P)...))) \rightarrowtail \lambda x_1\, x_2 \ldots x_n.P$$
$$((...(P_1\,P_2)...)\,P_n) \rightarrowtail P_1\,P_2 \ldots P_n$$

Esto nos permite escribir al combinador $Y$, de manera más breve como

$$\lambda f.(\lambda x.f(x\,x))\lambda x.f(x\,x)$$

**Sintaxis del lenguaje** Para definir de manera rigurosa las cadenas de símbolos que son elementos de $\Omega$ y el abuso de notación se implementa un *analizador sintáctico* (o *parser*) y un *sintetizador sintáctico* (o *escritor*).

La tarea del parser consiste en transformar una cadena de símbolos en su estructrura correspondiente, en esta implementación los símbolos son caracteres leídos de un puerto de entrada el cuál puede extraer la información de un archivo en disco o de una cadena de caracteres. Ya que no todos los lenguajes de programación tienen la capacidad de manejar letras griegas como '$\lambda$' se considera el caracter '\' como un reemplazo de la letra lambda. Si se dispone de una codificación de caracteres extendida como Unicode, entonces el caracter lambda (con código 955) puede ser utilizado también.

La tarea del escritor consiste en transformar una estructura a su cadena de símbolos correspondiente, ya que la cadena de símbolos puede o no usar el abuso de notación, se implementan dos algoritmos de escritura: uno formal y otro breve.

Una consideración del lenguaje de Lambda que no es explícitamente mencionada en la definición de arriba es que el símbolo '`;`' es utilizado para separar expresiones.

La implementación de la sintaxis del lenguaje tiene la siguiente forma:

13a ⟨*Sintaxis del lenguaje* 13a⟩≡ (32b)
⟨*Configuración de la sintaxis* 13b⟩
⟨*Parseo del lenguaje* 13c⟩
⟨*Escritura del lenguaje* 22⟩

**Configuración de la sintaxis** Tanto el parser como el escritor deben de poder manejar el caso en que la codificación del archivo o cadena a leer o escribir no soporte Unicode, la variable hay-lambda? debe de tener un valor booleano que indique si se puede leer y escribir el caracter '$\lambda$' con código 955. De igual manera la variable símbolo-lambda debe ser el valor del caracter lambda.

13b ⟨*Configuración de la sintaxis* 13b⟩≡ (13a)
```
(define hay-lambda? true)

(define/contract (símbolo-lambda? x)
  (char? . -> . boolean?)
  (and hay-lambda? (char=? (integer->char 955) x)))
```
Define:
hay-lambda?, utilizado en fragmentos 24c y 29.
símbolo-lambda?, utilizado en fragmentos 18d y 19b.

### 2.2.1 Analizador sintáctico

**Parseo del lenguaje** El parser es implementado con un algorimo *descendente recursivo* el cuál toma como argumento un puerto de entrada de Racket y construye la estructura correspondiente a la cadena de símbolos leída. La estructura de la implementación del parser es la siguiente:

13c ⟨*Parseo del lenguaje* 13c⟩≡ (13a)
⟨*Procedimientos auxiliares del parser* 15c⟩
⟨*Algoritmo del parser* 13d⟩
⟨*Punto de entrada del parser* 14a⟩

El procedimiento que implementa el algoritmo para analizar los caracteres del puerto de entrada es parsear-expresión, este procedimiento recibe un puerto de entrada p, un caracter c (que usualmente es el primer caracter de p) y un acumulador de expresiones a cuyo valor es la expresión parcialmente parseada; el valor resultante es una expresión en caso de éxito, un objeto "fin de archivo" cuando no hay símbolos que leer del puerto o un caracter cuando no se pudo parsear una expresión.

13d ⟨*Algoritmo del parser* 13d⟩≡ (13c)
```
(define/contract (parsear-expresión p c a)
  (input-port? (or/c char? eof-object?) (or/c expresión? null?)
               . -> . (or/c expresión? eof-object? char?))
  (cond ⟨Casos del algoritmo de parseo 14c⟩))
```
Define:
parsear-expresión, utilizado en fragmentos 14–20.
Utiliza expresión? 3a y parsear 14a.

El procedimiento parsear es el punto de entrada del parser, se encarga de intentar parsear una expresión utilizando el procedimiento parsear-expresión y emitir un error en caso que los símbolos del puerto no hayan podido conformar un elemento de Ω válido.

14a ⟨*Punto de entrada del parser* 14a⟩≡ (13c) 14b▸
```
(define/contract (parsear p)
  (input-port? . -> . (or/c expresión? eof-object?))
  (let ([e (parsear-expresión p (peek-char p) null)])
    (cond [(eof-object? e) e]
          [(expresión? e)  e]
          [(char=? #\; e)  (parsear p)]
          [else
           (error 'parsear "No fue posible parsear una expresión válida")])))
```
Define:
parsear, utilizado en fragmentos 13–21, 25b, 30–32, 35, 51, y 59b.
Utiliza expresión? 3a y parsear-expresión 13d.

Un procedimiento que resulta útil para hacer pruebas simples de parseo es parsear-cadena el cuál se encarga de convertir una cadena de caracteres en un puerto de entrada y parsear el puerto de entrada.

14b ⟨*Punto de entrada del parser* 14a⟩+≡ (13c) ◂14a
```
(define/contract (parsear-cadena s)
  (string? . -> . (or/c expresión? eof-object?))
  (parsear (open-input-string s)))
```
Define:
parsear-cadena, utilizado en fragmentos 15, 16, 21, 25b, 30–32, 35, 51, y 59b.
Utiliza expresión? 3a y parsear 14a.

**Casos del algoritmo de parseo** El procedimiento parsear-expresión comienza conociendo el primer caracter c del puerto de entrada p, se realiza un análisis de casos con cond para determinar la estructura sintáctica de la que c forma parte tomando en cuenta el parseo parcial acumulado a.

Antes de abordar cada caso es importante precisar el rol que juega el acumulador a. Ya que el parser debe ser capaz de construír una estructura a partir de una representación simbólica formal, parcialmente formal o con la mayor cantidad de abuso de notación posible, es necesario "recordar" la forma de las aplicaciones que se han leído, por ejemplo una expresión de la forma $(P_1\ P_2\ P_3)$ es formalmente $((P_1\ P_2)\ P_3)$, sin embargo, la lectura de izquierda a derecha nos obliga a leer primero $P_1$, acumular su estructura y al ver que es posible leer a $P_2$ construír la aplicación $(P_1\ P_2)$ y acumularla para intentar leer la siguiente expresión $P_3$ y formar la aplicación final.

**Cuando se termina de parsear** Hay dos situaciones en donde se debe terminar de parsear una expresión, la primera es cuando se lee un objeto "fin de archivo" y la otra es cuando se lee un *delimitador final*.

Si c satisface el predicado eof-object? es cuando ya no es posible extraer más caracteres de p.

14c ⟨*Casos del algoritmo de parseo* 14c⟩≡ (13d) 15e▸
```
[(eof-object? c)
 ⟨Se leyó un eof-object 15a⟩]
```

En este caso se debe regresar este objeto "fin de archivo" a menos que el valor del acumulador a no sea nulo, cuando a no es nulo es una expresión correctamente parseada, por lo tanto debe ser el resultado del parseo.

15a ⟨*Se leyó un* `eof-object` 15a⟩≡ (14c)
```
(if (null? a) c a)
```

El caso de prueba "`Lectura de fin de archivo`" verifica que este caso se maneje correctamente.

15b ⟨*Casos de prueba* 4a⟩+≡ (33) ◂10c 15d▸
```
(test-case "Lectura de fin de archivo"
  (check-equal? (parsear-cadena "") eof)
  (check-equal? (parsear-expresión (open-input-string "") eof (variable "x"))
                (variable "x")))
```
Utiliza `parsear` 14a, `parsear-cadena` 14b, `parsear-expresión` 13d, y `variable` 3b.

En caso que el parser se encuentre un delimitador final el algoritmo debe regresar, los delimitadores finales son aquellos objetos que delimitan expresiones por la derecha. El predicado `delimitador-final?` determina si un objeto es delimitador final.

15c ⟨*Procedimientos auxiliares del parser* 15c⟩≡ (13c) 17a▸
```
(define/contract (delimitador-final? x)
  ((or/c char? eof-object?) . -> . boolean?)
  (or (eof-object? x) (char=? #\] x) (char=? #\) x)
      (char=? #\. x)  (char=? #\, x) (char=? #\; x)))
```
Define:
`delimitador-final?`, utilizado en fragmentos 15 y 19b.

El caso de prueba "`Predicado para delimitadores`" verifica que este procedimiento sea correcto.

15d ⟨*Casos de prueba* 4a⟩+≡ (33) ◂15b 16a▸
```
(test-case "Predicados para delimitadores"
  (check-true (delimitador-final? eof))
  (check-true (delimitador-final? #\)))
  (check-false (delimitador-final? #\x))
  (check-false (delimitador-final? #\())
  (check-exn exn:fail:contract? (lambda () (delimitador-final? 42))))
```
Utiliza `delimitador-final?` 15c.

Si `c` satisface el predicado `delimitador-final?` el algoritmo ha terminado de parsear.

15e ⟨*Casos del algoritmo de parseo* 14c⟩+≡ (13d) ◂14c 16b▸
```
[(delimitador-final? c)
 ⟨Se leyó un delimitador final 15f⟩]
```
Utiliza `delimitador-final?` 15c.

De manera similar al caso anterior el resultado depende de `a`, cuando es nulo se debe consumir el delimitador del puerto de entrada y regresarse como resultado, de lo contrario, el delimitador debe permanecer en `p` y el resultado es la expresión acumulada.

15f ⟨*Se leyó un delimitador final* 15f⟩≡ (15e)
```
(if (null? a) (read-char p) a)
```

El caso de prueba "Lectura de delimitador final" verifica que este caso se maneje correctamente.

16a ⟨*Casos de prueba* 4a⟩+≡ (33) ◃15d 16d▹
```
(test-case "Lectura de delimitador final"
  (check-exn exn:fail? (lambda () (parsear-cadena ")")))
  (check-equal? (parsear-expresión (open-input-string ")") #\) null) #\))
  (let ([p (open-input-string ")")])
    (check-equal? (parsear-expresión p #\) (variable "x")) (variable "x"))
    (check-equal? (read-char p) #\)))
```
Utiliza parsear 14a, parsear-cadena 14b, parsear-expresión 13d, y variable 3b.

**Cuando se lee espacio en blanco** Los espacios en blanco, también conocidos como caracteres "whitespace" son ignorador por el parser. La implementación de este caso hace uso del predicado char-whitespace? de Racket para determinar si c es un espacio en blanco, este predicado reconoce al menos los caracteres espacio, tab y nueva línea.

16b ⟨*Casos del algoritmo de parseo* 14c⟩+≡ (13d) ◃15e 16e▹
```
[(char-whitespace? c)
 ⟨Se leyó un espacio en blanco 16c⟩]
```

Para ignorar a c y seguir parseando expresiones se debe consumir el caracter del puerto p e invocar de manera recursiva parsear-expresión.

16c ⟨*Se leyó un espacio en blanco* 16c⟩≡ (16b)
```
(read-char p)
(parsear-expresión p (peek-char p) a)
```
Utiliza parsear 14a y parsear-expresión 13d.

El caso de prueba "Lectura de espacio en blanco" verifica que este caso se maneje correctamente.

16d ⟨*Casos de prueba* 4a⟩+≡ (33) ◃16a 17c▹
```
(test-case "Lectura de espacio en blanco"
  (check-equal? (parsear-cadena " \t\n \n\t")
                (parsear-cadena ""))
  (check-equal? (parsear-expresión (open-input-string " \t\n \n\t)")
                                   #\space
                                   (variable "x"))
                (parsear-expresión (open-input-string ")") #\) (variable "x"))))
```
Utiliza parsear 14a, parsear-cadena 14b, parsear-expresión 13d, y variable 3b.

**Cuando se parsea una subexpresión** En el análisis de casos inicial del algoritmo, la única manera de comenzar a parsear una subexpresión es leyendo el caracter de paréntesis abierto.

16e ⟨*Casos del algoritmo de parseo* 14c⟩+≡ (13d) ◃16b 18a▹
```
[(char=? #\( c)
 ⟨Se leyó un paréntesis abierto 17d⟩]
```

En este caso se debe consumir el paréntesis abierto del puerto p, obtener la expresión dentro del paréntesis con un procedimiento auxiliar parsear-paréntesis e invocar de manera recursiva a parsear-expresión.

El procedimiento parsear-paréntesis recibe un puerto de entrada e invoca a parsear-expresión, eventualmente el parseo termina al encontrar un paréntesis cerrado, en cuyo caso se debe verificar que la expresión leída sea una abstracción o una aplicación, ya que estas son las únicas expresiones que pueden ir dentro de un paréntesis.

17a ⟨*Procedimientos auxiliares del parser* 15c⟩+≡ (13c) ◂15c 17b▸

```
(define/contract (parsear-paréntesis p)
  (input-port? . -> . (or/c abstracción? aplicación?))
  (let ([e (parsear-expresión p (peek-char p) null)])
    (unless (char=? #\) (read-char p))
      (error 'parsear-paréntesis "Se esperaba leer el caracter ‘)’"))
    e))
```

Define:
parsear-paréntesis, utilizado en fragmento 17d.
Utiliza abstracción 4c, aplicación 5a, parsear 14a, y parsear-expresión 13d.

La invocación recursiva a parsear-expresión debe de realizarse con cuidado, considerando a e la expresión resultante de parsear-paréntesis, si el valor del acumulador es nulo, la llamada recursiva se hace con e como acumulador, de lo contrario se deberá acumular la aplicación de a con e. Esta verificación se realiza con el procedimiento auxiliar aplicación/identidad.

17b ⟨*Procedimientos auxiliares del parser* 15c⟩+≡ (13c) ◂17a 18b▸

```
(define/contract (aplicación/identidad a e)
  ((or/c expresión? null?) expresión? . -> . expresión?)
  (if (null? a) e (aplicación a e)))
```

Define:
aplicación/identidad, utilizado en fragmentos 17, 18, y 20c.
Utiliza aplicación 5a y expresión? 3a.

El caso de prueba “Acumulación de expresiones” se encarga de verificar el correcto funcionamiento de este procedimiento.

17c ⟨*Casos de prueba* 4a⟩+≡ (33) ◂16d 21a▸

```
(test-case "Acumulación de expresiones"
  (check-equal? (aplicación/identidad null (variable "x"))
                (variable "x"))
  (check-equal? (aplicación/identidad (variable "x") (variable "y"))
                (aplicación (variable "x") (variable "y"))))
```

Utiliza aplicación 5a, aplicación/identidad 17b, y variable 3b.

Con estos procedimientos auxiliares implementados se define el caso de leer un paréntesis abierto.

17d ⟨*Se leyó un paréntesis abierto* 17d⟩≡ (16e)

```
(read-char p)
(let ([e (parsear-paréntesis p)])
  (parsear-expresión p (peek-char p) (aplicación/identidad a e)))
```

Utiliza aplicación 5a, aplicación/identidad 17b, parsear 14a, parsear-expresión 13d, y parsear-paréntesis 17a.

**Cuando se parsea un hueco** En el análisis de casos inicial del algoritmo, la única manera de comenzar a parsear un hueco es leyendo el caracter de corchete abierto, este caracter también forma parte de la representación simbólica de las metainstrucciones, sin embargo, en el caso de las metainstrucciones un nombre tiene que ser leído antes del caracter '`[`'.

18a ⟨*Casos del algoritmo de parseo* 14c⟩+≡ (13d) ◂16e 18d▸
```
[(char=? #\[ c)
 ⟨Se leyó un corchete abierto 18c⟩]
```

En este caso se hace uso del procedimiento auxiliar `parsear-hueco`, el cuál admite la lectura de cero o más espacios en blanco antes de encontrarse con un corchete cerrado.

18b ⟨*Procedimientos auxiliares del parser* 15c⟩+≡ (13c) ◂17b 19a▸
```
(define/contract (parsear-hueco p)
  (input-port? . -> . hueco?)
  (let ([c (peek-char p)])
    (cond [(char-whitespace? c)
           (read-char p)
           (parsear-hueco p)]
          [(char=? #\] c)
           (read-char p)
           (hueco)]
          [else
           (error 'parsear-hueco "Se esperaba leer el caracter ']'")])))
```
Define:
  `parsear-hueco`, utilizado en fragmento 18c.
Utiliza `hueco` 5c y `parsear` 14a.

La implementación del caso consume el caracter '`[`' del puerto `p` y acumula el hueco parseado con `parsear-hueco` en una invocación recursiva a `parsear-expresión`.

18c ⟨*Se leyó un corchete abierto* 18c⟩≡ (18a)
```
(read-char p)
(let ([e (parsear-hueco p)])
  (parsear-expresión p (peek-char p) (aplicación/identidad a e)))
```
Utiliza `aplicación` 5a, `aplicación/identidad` 17b, `hueco` 5c, `parsear` 14a, `parsear-expresión` 13d, y `parsear-hueco` 18b.

**Cuando se lee una lambda** En la representación simbólica de las abstracciones, los paréntesis externos pueden omitirse por abuso de notación, por lo que un símbolo inicial válido para una abstracción es el caracter '$\lambda$'.

18d ⟨*Casos del algoritmo de parseo* 14c⟩+≡ (13d) ◂18a 20a▸
```
[(or (símbolo-lambda? c) (char=? #\\ c))
 ⟨Se leyó una lambda 18e⟩]
```
Utiliza `símbolo-lambda?` 13b.

La implementación de este caso es similar a los anteriores dos, se consume el caracter leído, se utiliza un procedimiento auxiliar para parsear la abstracción y se invoca de manera recursiva `parsear-expresión` acumulando la abstracción resultante.

18e ⟨*Se leyó una lambda* 18e⟩≡ (18d)
```
(read-char p)
(let ([e (parsear-abstracción p)])
  (parsear-expresión p (peek-char p) (aplicación/identidad a e)))
```
Utiliza `abstracción` 4c, `aplicación` 5a, `aplicación/identidad` 17b, `parsear` 14a, `parsear-abstracción` 19a, y `parsear-expresión` 13d.

El procedimiento auxiliar parsear-abstracción es un poco más complejo que los anteriores procedimientos auxiliares. Realiza una iteración acumulando las variables leídas cuidando los casos en que se lea un delimitador antes que el caracter '.' que separa los argumentos del cuerpo e ignorando los espacios en blanco de variable a variable. Cuando el punto separador es leído, se parsea el cuerpo de la abstracción y se construye la estructura completa aplicando un algoritmo de "currificación" con las variables acumuladas.

19a ⟨*Procedimientos auxiliares del parser* 15c⟩+≡ (13c) ◂18b 19b▸

```
(define/contract (parsear-abstracción p)
  (input-port? . -> . abstracción?)
  (let itera ([c  (peek-char p)]
              [vs null])
    (cond [(and (char? c) (char-whitespace? c))
           (read-char p)
           (itera (peek-char p) vs)]
          [(and (char? c) (char=? #\. c))
           (read-char p)
           (if (null? vs)
               (error 'parsear-abstracción "Se esperaba al menos una variable")
               (let ([e (parsear-expresión p (peek-char p) null)])
                 (if (expresión? e)
                     (foldr abstracción e (reverse vs))
                     (error 'parsear-abstracción
                            "Se esperaba leer el cuerpo de una abstracción"))))]
          [(delimitador? c)
           (error 'parsear-abstracción "Se esperaba leer una abstracción completa")]
          [else
           (let ([v (variable (parsear-nombre p))])
             (itera (peek-char p) (cons v vs)))])))
```

Define:
parsear-abstracción, utilizado en fragmento 18e.
Utiliza abstracción 4c, delimitador? 19b, expresión? 3a, parsear 14a, parsear-expresión 13d, parsear-nombre 20b, y variable 3b.

El predicado delimitador? es implementado de manera similar a delimitador-final? solo que contempla todos los objetos que delimitan expresiones, incluyendo los delimitadores ignorados de espacios en blanco.

19b ⟨*Procedimientos auxiliares del parser* 15c⟩+≡ (13c) ◂19a 20b▸

```
(define/contract (delimitador? x)
  ((or/c char? eof-object?) . -> . boolean?)
  (or (delimitador-final? x)
      (char=? #\( x) (char=? #\[ x)
      (símbolo-lambda? x) (char=? #\\ x)
      (char-whitespace? x)))
```

Define:
delimitador?, utilizado en fragmentos 19a y 20b.
Utiliza delimitador-final? 15c y símbolo-lambda? 13b.

En el parseo de abstracciones se hace uso del procedimiento parsear-nombre, este es descrito en el siguiente caso.

**Cuando se lee un nombre** El último caso del análisis de casos es leer un caracter que no es delimitador, lo cuál significa que constituye un nombre.

20a ⟨*Casos del algoritmo de parseo* 14c⟩+≡ (13d) ◂18d
```
[else
 ⟨Se leyó un constituyente 20c⟩]
```

Para leer un nombre se consumen y acumulan caracteres constituyentes (no-delimitadores) hasta encontrar algún otro caracter. El procedimiento `parsear-nombre` regresa una cadena con los caracteres consumidos del puerto `p`.

20b ⟨*Procedimientos auxiliares del parser* 15c⟩+≡ (13c) ◂19b 20d▸
```
(define/contract (parsear-nombre p)
  (input-port? . -> . string?)
  (let itera ([c  (peek-char p)]
              [cs null])
    (if (delimitador? c)
        (if (null? cs)
            (error 'parsear-nombre "Se intentó leer un nombre de longitud cero")
            (list->string (reverse cs)))
        (begin
         (read-char p)
         (itera (peek-char p) (cons c cs))))))
```
Define:
`parsear-nombre`, utilizado en fragmentos 19a y 20c.
Utiliza `delimitador?` 19b, `longitud` 57a, y `parsear` 14a.

En la implementación del caso, se parsea un nombre y se determina el tipo de resultado, cuando el caracter después del nombre es '`[`' el resultado es una metainstrucción, de lo contrario es una variable.

20c ⟨*Se leyó un constituyente* 20c⟩≡ (20a)
```
((lambda (e)
   (parsear-expresión p (peek-char p) (aplicación/identidad a e)))
 (let* ([x (parsear-nombre p)]
        [c (peek-char p)])
   (cond [(and (char? c) (char=? #\[ c))
          (read-char p)
          (metainstrucción x (parsear-corchetes p))]
         [else
          (variable x)])))
```
Utiliza `aplicación` 5a, `aplicación/identidad` 17b, `metainstrucción` 6a, `parsear` 14a, `parsear-corchetes` 20d, `parsear-expresión` 13d, `parsear-nombre` 20b, y `variable` 3b.

En el caso que el nombre sea de una metainstrucción, se parsea la lista de argumentos con el procedimiento auxiliar `parsear-corchetes`.

20d ⟨*Procedimientos auxiliares del parser* 15c⟩+≡ (13c) ◂20b
```
(define/contract (parsear-corchetes p)
  (input-port? . -> . (*list/c expresión?))
  (let itera ([e  (parsear-expresión p (peek-char p) null)]
              [es null])
    (cond [(eof-object? e)
           (error 'parsear-corchetes
                  "Se esperaba leer una metainstrucción completa")]
          [(char? e)
           (case e
             [(#\]) (reverse es)]
             [(#\,) (itera (parsear-expresión p (peek-char p) null) es)]
```

```
            [else
             (error 'parsear-corchetes
                    "Se esperaba leer una metainstrucción completa")])]
          [else
           (itera (parsear-expresión p (peek-char p) null)
                  (cons e es))])))
```

Define:
parsear-corchetes, utilizado en fragmento 20c.
Utiliza expresión? 3a, metainstrucción 6a, parsear 14a, y parsear-expresión 13d.

Para verificar el correcto funcionamiento del parser se definen dos casos de prueba para el parseo de representaciones simbólicas con y sin abuso de notación.

21a ⟨*Casos de prueba* 4a⟩+≡ (33) ◂17c 21b▸

```
(test-case "Parsear sin abuso de notación"
  (check-equal? (parsear-cadena "foo")
                (variable "foo"))
  (check-equal? (parsear-cadena "(foo bar)")
                (aplicación (variable "foo") (variable "bar")))
  (check-equal? (parsear-cadena "((foo bar) baz)")
                (aplicación (aplicación (variable "foo") (variable "bar"))
                            (variable "baz")))
  (check-equal? (parsear-cadena "(foo (bar baz))")
                (aplicación (variable "foo")
                            (aplicación (variable "bar") (variable "baz"))))
  (check-equal? (parsear-cadena "(\\bar.bar)")
                (abstracción (variable "bar") (variable "bar")))
  (check-equal? (parsear-cadena "foo[bar,baz]")
                (metainstrucción "foo" (list (variable "bar") (variable "baz"))))
  (check-equal? (parsear-cadena "(\\foo.(\\bar.foo))")
                (abstracción (variable "foo")
                             (abstracción (variable "bar") (variable "foo"))))
  (check-equal? (parsear-cadena "  []  ")
                (hueco)))
```

Utiliza abstracción 4c, aplicación 5a, hueco 5c, metainstrucción 6a, parsear 14a, parsear-cadena 14b, y variable 3b.

El caso de prueba para parsear expresiones escritas con abuso de notación debe ser más exhaustivo ya que el abuso de notación puede ser parcial o total.

21b ⟨*Casos de prueba* 4a⟩+≡ (33) ◂21a 25b▸

```
(test-case "Parsear con abuso de notación"
  (check-equal? (parsear-cadena "foo bar")
                (aplicación (variable "foo") (variable "bar")))
  (check-equal? (parsear-cadena "foo bar baz")
                (aplicación (aplicación (variable "foo") (variable "bar"))
                            (variable "baz")))
  (check-equal? (parsear-cadena "foo (bar baz)")
                (aplicación (variable "foo")
                            (aplicación (variable "bar") (variable "baz"))))
  (check-equal? (parsear-cadena "\\foo.foo")
                (abstracción (variable "foo") (variable "foo")))
  (check-equal? (parsear-cadena "\\foo bar.foo")
                (abstracción (variable "foo")
                             (abstracción (variable "bar") (variable "foo"))))
  (check-equal? (parsear-cadena "(\\foo bar.foo)")
```

```
                (abstracción (variable "foo")
                             (abstracción (variable "bar") (variable "foo"))))
    (check-equal? (parsear-cadena "(\\foo.\\bar.foo)")
                (abstracción (variable "foo")
                             (abstracción (variable "bar")
                                          (variable "foo"))))
    (check-equal? (parsear-cadena "\\foo.bar baz")
                (abstracción (variable "foo")
                             (aplicación (variable "bar")
                                         (variable "baz"))))
    (check-equal? (parsear-cadena "\\foo bar.bar baz")
                (abstracción (variable "foo")
                             (abstracción (variable "bar")
                                          (aplicación (variable "bar")
                                                      (variable "baz")))))
    (check-equal? (parsear-cadena "(\\foo.foo)\\bar.bar")
                (aplicación (abstracción (variable "foo")
                                         (variable "foo"))
                            (abstracción (variable "bar")
                                         (variable "bar"))))
    (check-equal? (parsear-cadena "foo[\\bar.bar bar, \\baz.baz baz] quux")
                (aplicación
                 (metainstrucción "foo"
                                  (list (abstracción (variable "bar")
                                                     (aplicación (variable "bar")
                                                                 (variable "bar")))
                                        (abstracción (variable "baz")
                                                     (aplicación (variable "baz")
                                                                 (variable "baz")))))
                 (variable "quux")))
    (check-equal? (parsear-cadena "\\x.x [] x")
                (abstracción (variable "x")
                             (aplicación (aplicación (variable "x") (hueco))
                                         (variable "x")))))
```

Utiliza abstracción 4c, aplicación 5a, hueco 5c, metainstrucción 6a, parsear 14a, parsear-cadena 14b, y variable 3b.

### 2.2.2 Sintetizador sintáctico

**Escritura del lenguaje** Para escribir estructuras de expresiones se identifican dos algoritmos básicos: uno para escritura formal y otro para escritura breve. Estos algoritmos se basan en un procedimiento de *recorrido a lo profundo* en *postorden* para transformar la estructura de una expresión a otro tipo de objetos, como por ejemplo cadenas de caracteres. Es importante implementar estos algoritmos de la manera más general posible, esto con la finalidad de poder definir escritores para diversos "back-ends" como cadenas de caracteres en texto plano o código del modo de matemáticas de LaTeX. La estructura de esta implementación es la siguiente:

22 ⟨*Escritura del lenguaje* 22⟩≡ (13a)
  ⟨*Algoritmos principales de escritura* 23a⟩
  ⟨*Escritores usuales* 24a⟩
  ⟨*Escritores misceláneos* 25c⟩

**Constructor de escritores formales** La técnica utilizada para implementar estos algoritmos genéricos es definir un constructor de escritores que reciba como argumentos procedimientos específicos para escribir cada clase de estructura. Consideremos primero el constructor de escritores formales llamado `escritor-formal`, toma cinco procedimientos como argumentos, uno por cada tipo de estructura y regresa un escritor concreto.

23a ⟨*Algoritmos principales de escritura* 23a⟩≡ (22) 27▸
```
(define/contract (escritor-formal e-variable e-abstracción e-aplicación
                                  e-hueco e-metainstrucción)
  ⟨Contrato para escritores formales 23b⟩
  ⟨Algoritmo de escritura formal 23c⟩)
```
Define:
escritor-formal, utilizado en fragmentos 24–26 y 47.
Utiliza abstracción 4c, aplicación 5a, hueco 5c, metainstrucción 6a, y variable 3b.

El contrato de este constructor es que recibe cinco procedimientos y regresa un procedimiento que recibe una expresión y regresa algún objeto. Cada procedimiento de argumento corresponde en orden a escritor de variables, abstracciones, aplicaciones, huecos y metainstrucciones de tal manera que cada uno de estos procedimientos recibe de argumento la esctructura de la expresión correspondiente y los campos de dicha estructura.

23b ⟨*Contrato para escritores formales* 23b⟩≡ (23a)
```
((variable? string? . -> . any/c)                 ; Escritor de variables
 (abstracción? variable? expresión? . -> . any/c) ; Escritor de abstracciones
 (aplicación? expresión? expresión? . -> . any/c) ; Escritor de aplicaciones
 (hueco? . -> . any/c)                            ; Escritor de huecos
 (metainstrucción? string? (*list/c expresión?)   ; Escritor de metainstrucciones
                   . -> . any/c)
 . -> . (expresión? . -> . any/c))      ; Escritor de expresiones especializado
```
Utiliza abstracción 4c, aplicación 5a, expresión? 3a, hueco 5c, metainstrucciones 6c, metainstrucción 6a, y variable 3b.

El algoritmo en sí tiene la tarea de identificar el escritor especializado que invocar y extraer los campos de la expresión e.

23c ⟨*Algoritmo de escritura formal* 23c⟩≡ (23a)
```
(lambda (e)
  (cond [(variable? e)
         (e-variable e (variable-nombre e))]
        [(abstracción? e)
         (e-abstracción e (abstracción-argumento e) (abstracción-cuerpo e))]
        [(aplicación? e)
         (e-aplicación e (aplicación-operador e) (aplicación-operando e))]
        [(hueco? e)
         (e-hueco e)]
        [(metainstrucción? e)
         (e-metainstrucción e (metainstrucción-nombre e)
                            (metainstrucción-argumentos e))]))
```
Utiliza abstracción 4c, aplicación 5a, hueco 5c, metainstrucción 6a, y variable 3b.

**Escritor formal a texto plano** La definición de un escritor formal a una cadena de caracteres en texto plano es relativamente fácil. El objeto que regresan los procedimientos

especializados es una cadena de caracteres y la escritura de estructuras complejas consiste en concatenar los resultados de la escritura de sus partes.

24a ⟨*Escritores usuales* 24a⟩≡ (22) 29▸
```
(define escribir-expresión-formal
  (escritor-formal
   ⟨Escribir formalmente variable a texto plano 24b⟩
   ⟨Escribir formalmente abstracción a texto plano 24c⟩
   ⟨Escribir formalmente aplicación a texto plano 24d⟩
   ⟨Escribir formalmente hueco a texto plano 24e⟩
   ⟨Escribir formalmente metainstrucción a texto plano 25a⟩))
```
Define:
escribir-expresión-formal, utilizado en fragmentos 24, 25, 35, y 45.
Utiliza escritor-formal 23a.

La representación simbólica de una variable es su nombre, ya que los nombres son internamente implementados como cadenas de caracteres es suficiente regresar este campo de la variable.

24b ⟨*Escribir formalmente variable a texto plano* 24b⟩≡ (24a)
```
(lambda (e nombre) nombre)
```

En el caso de las abstracciones se deben escribir de la forma $(\lambda x.M)$ donde $x$ es la representación simbólica del argumento y $M$ la representación simbólica del cuerpo. Para escribir el símbolo '$\lambda$' se debe verificar si se puede usar su código en Unicode, de lo contrario usar el caracter '\'.

24c ⟨*Escribir formalmente abstracción a texto plano* 24c⟩≡ (24a)
```
(lambda (e argumento cuerpo)
  (format "(~a~a.~a)"
          (if hay-lambda? (integer->char 955) #\\)
          (escribir-expresión-formal argumento)
          (escribir-expresión-formal cuerpo)))
```
Utiliza escribir-expresión-formal 24a y hay-lambda? 13b.

Las aplicaciones son escritas de manera similar, invocando escribir-expresión-formal de manera recursiva sobre sus campos.

24d ⟨*Escribir formalmente aplicación a texto plano* 24d⟩≡ (24a)
```
(lambda (e operador operando)
  (format "(~a ~a)"
          (escribir-expresión-formal operador)
          (escribir-expresión-formal operando)))
```
Utiliza escribir-expresión-formal 24a.

La representación simbólica de los huecos siempre es [ ], así que simplemente se regresa esta cadena de caracteres.

24e ⟨*Escribir formalmente hueco a texto plano* 24e⟩≡ (24a)
```
(lambda (e) "[ ]")
```

Para escribir una metainstrucción se obtienen las representaciones simbólicas de cada argumento y se concatenan junto con la separación de coma entre cada argumento.

25a ⟨*Escribir formalmente metainstrucción a texto plano* 25a⟩≡ (24a)
```
(lambda (e nombre argumentos)
  (format "~a[~a]" nombre
          (apply string-append
                 (add-between (map escribir-expresión-formal argumentos)
                              ", "))))
```
Utiliza escribir-expresión-formal 24a.

El caso de prueba "Escritura formal a texto plano" verifica que la implementación de este escritor concreto sea correcta.

25b ⟨*Casos de prueba* 4a⟩+≡ (33) ◂21b 30a▸
```
(test-case "Escritura formal a texto plano"
  (check-equal? (escribir-expresión-formal (parsear-cadena "foo"))
                "foo")
  (check-equal? (escribir-expresión-formal (parsear-cadena "\\foo.foo"))
                (format "(~afoo.foo)" (integer->char 955)))
  (check-equal? (escribir-expresión-formal
                 (parsear-cadena "foo bar (baz baz)"))
                "((foo bar) (baz baz))")
  (check-equal? (escribir-expresión-formal
                 (parsear-cadena "   [       ]"))
                "[ ]")
  (check-equal? (escribir-expresión-formal
                 (parsear-cadena "foo[\\bar.baz,baz bar]"))
                (format "foo[(~abar.baz), (baz bar)]" (integer->char 955))))
```
Utiliza escribir-expresión-formal 24a, parsear 14a, y parsear-cadena 14b.

**Escritor formal a LaTeX** La implementación básica del escritor formal a cadenas en texto plano es un buen punto de partida para definir escritores más especializados. Por ejemplo para escribir de manera formal a cadenas de caracteres con una sintáxis para el modo matemáticas de LaTeX se parte de la definición de escribir-expresión-formal para escribir '$\lambda$' como \lambda, espacios pequeños como \, y espacios grandes como \quad.

25c ⟨*Escritores misceláneos* 25c⟩≡ (22) 26a▸
```
(define escribir-latex-expresión-formal
  (escritor-formal
   (lambda (e nombre) nombre)
   (lambda (e argumento cuerpo)
     (format "(\\lambda ~a.~a)"
             (escribir-latex-expresión-formal argumento)
             (escribir-latex-expresión-formal cuerpo)))
   (lambda (e operador operando)
     (format "(~a\\, ~a)"
             (escribir-latex-expresión-formal operador)
             (escribir-latex-expresión-formal operando)))
   (lambda (e) "[\\quad ]")
   (lambda (e nombre argumentos)
     (format "~a[~a]" nombre
             (apply string-append
                    (add-between (map escribir-latex-expresión-formal
```

```
                argumentos)
            ",\\, ")))))
```

Define:
escribir-latex-expresión-formal, nunca utilizado.
Utiliza escritor-formal 23a.

Con el procedimiento escribir-latex-expresión-formal sólo hace falta usar una plantilla de documento de LaTeX para poder generar un archivo interpretable válido. Por ejemplo, la variable plantilla-latex tiene como valor una cadena de caracteres de tal manera que si e es una expresión y p es un puerto de salida a un archivo en disco
(fprintf p plantilla-latex e)
escribe al archivo un documento compilable de LaTeX.

26a ⟨*Escritores misceláneos* 25c⟩+≡ (22) ◂25c 26b▸

```
(define plantilla-latex
#<<DOCUMENTO-LATEX

\documentclass[preview]{standalone}
\usepackage{amsmath}

\begin{document}
\( ~a \)
\end{document}

DOCUMENTO-LATEX
  )
```

Define:
plantilla-latex, nunca utilizado.

**Escritura a otros lenguajes** Traducir expresiones a LaTeX puede ser de utilidad para leer expresiones con tipografía más bonita. Sin embargo el algoritmo de escritura también puede ser utilizado para implementar escritores con poco valor estético pero grán valor práctico. Por ejemplo para escribir expresiones a JSON con la finalidad de transferir una expresión particular a un lenguaje con la habilidad de parsear este popular formato, se puede utilizar el procedimiento escribir-json-expresión-formal.

26b ⟨*Escritores misceláneos* 25c⟩+≡ (22) ◂26a 30b▸

```
(define escribir-json-expresión-formal
  (escritor-formal
   (lambda (e nombre)
     (format "{ ~s : ~s, ~s : ~s }"
             "tipo" "variable"
             "nombre" nombre))
   (lambda (e argumento cuerpo)
     (format "{ ~s : ~s, ~s : ~a, ~s : ~a }"
             "tipo" "abstracción"
             "argumento" (escribir-json-expresión-formal argumento)
             "cuerpo" (escribir-json-expresión-formal argumento)))
   (lambda (e operador operando)
     (format "{ ~s : ~s, ~s : ~a, ~s : ~a }"
             "tipo" "aplicación"
             "operador" (escribir-json-expresión-formal operador)
             "operando" (escribir-json-expresión-formal operando)))
```

```
(lambda (e)
  (format "{ ~s : ~s }"
          "tipo" "hueco"))
(lambda (e nombre argumentos)
  (format "{ ~s : ~s, ~s : ~s, ~s : [~a] }"
          "tipo" "metainstrucción"
          "nombre" nombre
          "argumentos"
          (apply string-append
                 (add-between (map escribir-json-expresión-formal
                                   argumentos)
                              ", "))))))
```

Define:
escribir-json-expresión-formal, nunca utilizado.
Utiliza abstracción 4c, aplicación 5a, escritor-formal 23a, hueco 5c, metainstrucción 6a, y variable 3b.

**Constructor de escritores breves** El algoritmo complicado de escritura es el que incluye abuso de notación. El parseo de expresiones admite abuso de notación parcial o total y para cualquier entrada se tiene una única salida, por otro lado en el caso de la escritura de expresiones con abuso, se tienen muchas posibles salidas para una misma entrada. En lambda se considera la escritura con abuso total de notación, es decir, si es posible aplicar un abuso de notación, se aplica.

De manera similar al constructor escritor-formal se implementa otro constructor llamado escritor-breve el cuál toma también procedimientos de escritura especializados para cada tipo de estructura. Los procedimientos especializados para las abstracciones y las aplicaciones deben satisfacer contratos diferentes a los de la escritura formal ya que estos dos tipos de expresiones son las que tienen abuso de notación.

27 ⟨*Algoritmos principales de escritura* 23a⟩+≡ (22) ◃23a

```
(define/contract (escritor-breve e-variable e-abstracción e-aplicación
                                 e-hueco e-metainstrucción)
  ((variable? string? . -> . any/c)
   ⟨Contrato para escritor formal de abstracciones 28a⟩
   ⟨Contrato para escritor formal de aplicaciones 28b⟩
   (hueco? . -> . any/c)
   (metainstrucción? string? (*list/c expresión?) . -> . any/c)
   . -> . (expresión? . -> . any/c))
  ⟨Algoritmo de escritura breve 28c⟩)
```

Define:
escritor-breve, utilizado en fragmentos 29, 30b, 37, y 47.
Utiliza abstracción 4c, aplicación 5a, expresión? 3a, hueco 5c, metainstrucción 6a, y variable 3b.

Consideremos una abstracción de la forma

$$(\lambda x.(\lambda y.(\lambda z.M)))$$

tal que $M$ no es abstracción. El abuso de notación nos permite escribirla como

$$\lambda x\, y\, z.M$$

en donde los cambios realizados son la omisión de paréntesis externos y la "descurrificación" de argumentos en abstracciones anidadas (abstracciones cuyos cuerpos son abstracciones). Por lo tanto el procedimiento de escritura breve de abstracciones debe conocer la lista de

argumentos de la descurrificación (en este ejemplo $x$, $y$ y $z$) y el primer cuerpo en la anidación que no es abstracción (en este ejemplo $M$). La omisión de los paréntesis externos es la convención del abuso de notación, es posible que en una representación simbólica breve una abstracción esté encerrada entre paréntesis, sin embargo, la tarea de determinar si debe o no escribirse con paréntesis es del procedimiento de escritura de aplicaciones.

28a ⟨*Contrato para escritor formal de abstracciones* 28a⟩≡ (27)
```
(abstracción? (*list/c variable? variable?) expresión?
              . -> . any/c)
```
Utiliza abstracción 4c, expresión? 3a, y variable 3b.

En el caso de una aplicación de la forma

$$(((M\,N)\,O)\,P)$$

tal que $M$ no es aplicación. El abuso de notación nos permite escribirla como

$$M\,N\,O\,P$$

en donde los cambios realizados son la omisión de paréntesis externos, remover la asociación a la izquierda y determinar cuáles de las expresiones $M$, $N$, $O$ y $P$ deben de tener paréntesis. Este último procedimiento considera únicamente el caso en que una expresión de la aplicación sea abstracción y otra aplicación: para las abstracciones, todas excepto la última (en este ejemplo $P$) debe tener paréntesis, mientras que en el caso de las aplicaciones, todas deben tener paréntesis. El procedimiento de escritura breve para aplicaciones debe conocer la lista de expresiones en la aplicación y la lista de criterios que determinan qué expresiones deben tener paréntesis.

28b ⟨*Contrato para escritor formal de aplicaciones* 28b⟩≡ (27)
```
(aplicación? (*list/c expresión? expresión? expresión?)
             (*list/c boolean? boolean? boolean?)
             . -> . any/c)
```
Utiliza aplicación 5a y expresión? 3a.

El algoritmo de escritura breve funciona de manera similar al de escritura formal, solo que debe extraer la información necesaria para los escritores de abstracciones y aplicaciones.

28c ⟨*Algoritmo de escritura breve* 28c⟩≡ (27)
```
(lambda (e)
  (cond [(variable? e)
         (e-variable e (variable-nombre e))]
        [(abstracción? e)
         (let itera ([cuerpo (abstracción-cuerpo e)]
                     [argumentos (list (abstracción-argumento e))])
           (if (abstracción? cuerpo)
               (itera (abstracción-cuerpo cuerpo)
                      (cons (abstracción-argumento cuerpo)
                            argumentos))
               (e-abstracción e (reverse argumentos) cuerpo)))]
        [(aplicación? e)
         (let itera ([operador (aplicación-operador e)]
                     [operandos (list (aplicación-operando e))]
                     [paréntesis (list (aplicación? (aplicación-operando e)))])
           (if (aplicación? operador)
```

```
                (let ([operando (aplicación-operando operador)])
                  (itera (aplicación-operador operador)
                         (cons operando operandos)
                         (cons (or (aplicación? operando)
                                   (abstracción? operando))
                               paréntesis)))
                (e-aplicación e (cons operador operandos)
                              (cons (or (aplicación? operador)
                                        (abstracción? operador))
                                    paréntesis))))]
          [(hueco? e)
           (e-hueco e)]
          [(metainstrucción? e)
           (e-metainstrucción e (metainstrucción-nombre e)
                              (metainstrucción-argumentos e))]))
```

Utiliza abstracción 4c, aplicación 5a, hueco 5c, metainstrucción 6a, y variable 3b.

**Escritor breve a texto plano** La definición de un escritor breve a una cadena de caracteres en texto plano es relativamente fácil de implementar utilizando este algoritmo. Al igual que con `escribir-expresión-formal`, todos los escritores especializados regresan una cadena de caracteres.

29 ⟨*Escritores usuales* 24a⟩+≡ (22) ◃24a

```
(define escribir-expresión-breve
  (escritor-breve
   (lambda (e nombre) nombre)
   (lambda (e argumentos cuerpo)
     (format "~a~a.~a"
             (if hay-lambda? (integer->char 955) #\\)
             (apply string-append
                    (add-between (map escribir-expresión-breve argumentos)
                                 " "))
             (escribir-expresión-breve cuerpo)))
   (lambda (e operandos paréntesis)
     (apply string-append
            (add-between (map (lambda (o p)
                                (format "~a~a~a"
                                        (if p "(" "")
                                        (escribir-expresión-breve o)
                                        (if p ")" "")))
                              operandos paréntesis)
                         " ")))
   (lambda (e) "[ ]")
   (lambda (e nombre argumentos)
     (format "~a[~a]" nombre
             (apply string-append
                    (add-between (map escribir-expresión-breve argumentos)
                                 ", "))))))
```

Define:
`escribir-expresión-breve`, utilizado en fragmentos 30a y 45.
Utiliza `escritor-breve` 27 y `hay-lambda?` 13b.

El caso de prueba "`Escritura breve a texto plano`" verifica que la implementación de este escritor sea correcta.

30a ⟨*Casos de prueba* 4a⟩+≡ (33) ◂25b 32a▸
```
(test-case "Escritura breve a texto plano"
  (check-equal? (escribir-expresión-breve (parsear-cadena "foo"))
                "foo")
  (check-equal? (escribir-expresión-breve (parsear-cadena "\\foo.foo"))
                (format "~afoo.foo" (integer->char 955)))
  (check-equal? (escribir-expresión-breve
                 (parsear-cadena "foo bar (baz baz)"))
                "foo bar (baz baz)")
  (check-equal? (escribir-expresión-breve
                 (parsear-cadena "foo (bar baz) \\quux.quux quux"))
                (format "foo (bar baz) ~aquux.quux quux" (integer->char 955)))
  (check-equal? (escribir-expresión-breve (parsear-cadena " [    ]"))
                "[ ]")
  (check-equal? (escribir-expresión-breve
                 (parsear-cadena "foo[\\bar.baz,baz bar]"))
                (format "foo[~abar.baz, baz bar]" (integer->char 955))))
```
Utiliza `escribir-expresión-breve` 29, `parsear` 14a, y `parsear-cadena` 14b.

**Escritor breve a LaTeX** Para completar el ejemplo de escritura a LaTeX se presenta el escritor `escribir-latex-expresión-breve` utilizando el algoritmo que aplica abuso de notación.

30b ⟨*Escritores misceláneos* 25c⟩+≡ (22) ◂26b
```
(define escribir-latex-expresión-breve
  (escritor-breve
   (lambda (e nombre) nombre)
   (lambda (e argumentos cuerpo)
     (format "\\lambda ~a.~a"
             (apply string-append
                    (add-between (map escribir-latex-expresión-breve
                                      argumentos)
                                 "\\, "))
             (escribir-latex-expresión-breve cuerpo)))
   (lambda (e operandos paréntesis)
     (apply string-append
            (add-between (map (lambda (o p)
                                (format "~a~a~a"
                                        (if p "(" "")
                                        (escribir-latex-expresión-breve o)
                                        (if p ")" "")))
                              operandos paréntesis)
                         "\\, ")))
   (lambda (e) "[\\quad ]")
   (lambda (e nombre argumentos)
     (format "~a[~a]" nombre
             (apply string-append
                    (add-between (map escribir-latex-expresión-breve
                                      argumentos)
                                 ",\\, "))))))
```
Define:
`escribir-latex-expresión-breve`, nunca utilizado.

Utiliza escritor-breve 27.

## 2.3 Evaluación

**Evaluación de expresiones** Habiendo definido la sintaxis y las estructuras del lenguaje solo falta proveer un mecanismo para el análisis, manipulación y transformación de expresiones del lenguaje. Como se mencionó al implementar la estructura de las metainstrucciones, se registran procedimientos en una tabla hash llamada metainstrucciones los cuales esperan ser invocados con expresiones como argumentos.

La tarea de invocar metainstrucciones definidas la realiza un evaluador de expresiones, el algoritmo de evaluación consiste en recorrer las subexpresiones de una expresión en busca de invocaciones a metainstrucciones, al encontrar una metainstrucción se busca su nombre en el registro metainstrucciones, si su nombre es encontrado, se invoca su procedimiento asociado con las expresiones evaluadas de argumentos.

31a ⟨*Evaluación de expresiones* 31a⟩≡ (32b) 31b▸

```
(define/contract (evaluar-expresión e)
  (expresión? . -> . any)
  (cond [(abstracción? e)
         (abstracción (evaluar-expresión (abstracción-argumento e))
                      (evaluar-expresión (abstracción-cuerpo e)))]
        [(aplicación? e)
         (aplicación (evaluar-expresión (aplicación-operador e))
                     (evaluar-expresión (aplicación-operando e)))]
        [(metainstrucción? e)
         (cond [(hash-ref metainstrucciones (metainstrucción-nombre e) #f) =>
                (lambda (f)
                  (apply f (map evaluar-expresión (metainstrucción-argumentos e))))]
               [else
                (metainstrucción (metainstrucción-nombre e)
                                 (map evaluar-expresión
                                      (metainstrucción-argumentos e)))])]
        [else e]))
```

Define:
evaluar-expresión, utilizado en fragmentos 31b y 45.
Utiliza abstracción 4c, aplicación 5a, expresión? 3a, metainstrucciones 6c, y metainstrucción 6a.

De manera similar a la definición del procedimiento parsear-cadena se implementa evaluar-cadena para facilitar la invocación del evaluador.

31b ⟨*Evaluación de expresiones* 31a⟩+≡ (32b) ◂31a

```
(define/contract (evaluar-cadena s)
  (string? . -> . any)
  (evaluar-expresión (parsear-cadena s)))
```

Define:
evaluar-cadena, utilizado en fragmento 32a.
Utiliza evaluar-expresión 31a, parsear 14a, y parsear-cadena 14b.

Considerando metainstrucciones ficticias, el caso de prueba "Evaluación de expresiones" verifica que el evaluador funciona correctamente.

32a ⟨*Casos de prueba* 4a⟩+≡ (33) ◂30a

```
(define-metainstrucción (siempre-x . argumentos)
  (variable "x"))

(define-metainstrucción (encuentra-foo e)
  (define (busca-foo e)
    (cond [(or (variable? e) (hueco? e))
           #f]
          [(abstracción? e)
           (busca-foo (abstracción-cuerpo))]
          [(aplicación? e)
           (or (busca-foo (aplicación-operador e))
               (busca-foo (aplicación-operando e)))]
          [(metainstrucción? e)
           (if (string=? "foo" (metainstrucción-nombre e)) e #f)]))
  (let ([foo (busca-foo e)])
    (if foo foo e)))

(test-case "Evaluación de expresiones"
           (check-equal? (evaluar-cadena "y y siempre-x[y] y y")
                         (parsear-cadena "y y x y y"))
           (check-equal? (evaluar-cadena "\\x.encuentra-foo[x]")
                         (parsear-cadena "\\x.x"))
           (check-equal? (evaluar-cadena "\\x.encuentra-foo[foo[x]]")
                         (parsear-cadena "\\x.foo[x]"))
           (check-equal? (evaluar-cadena "siempre-x[encuentra-foo[foo[x]]]")
                         (parsear-cadena "x"))
           (check-equal? (evaluar-cadena "encuentra-foo[siempre-x[foo[x]]]")
                         (parsear-cadena "x")))
```

Utiliza abstracción 4c, aplicación 5a, define-metainstrucción 6c, evaluar-cadena 31b, hueco 5c, metainstrucción 6a, parsear 14a, parsear-cadena 14b, y variable 3b.

## 2.4 Código fuente

Los fragmentos de código presentados en este capítulo conforman un módulo de Racket llamado lambda-lib, este es dividido en dos partes: el archivo lambda-lib.rkt con la implementación del módulo y el archivo pruebas-lambda-lib.rkt el cuál contiene todos los casos de pruebas.

**Archivo** lambda-lib En este archivo se especifica el uso del lenguaje Racket y se proveen todas las definiciones presentes con (provide (all-defined-out)), el código principal de las tres secciones de este capítulo conforma el cuerpo del archivo, en orden.

32b ⟨*lambda-lib.rkt* 32b⟩≡

```
#lang racket
(provide (all-defined-out))
⟨Estructuras de las expresiones 11⟩
⟨Sintaxis del lenguaje 13a⟩
⟨Evaluación de expresiones 31a⟩
```

**Archivo** `pruebas-lambda-lib` Este archivo se conforma de todas las dependencias para los casos de prueba de este capítulo, una última dependencia es el archivo `lambda-lib.rkt` necesario para poder invocar los procedimientos definidos. El cuerpo de este archivo contiene todos los casos de prueba mencionados.

33 ⟨*pruebas-lambda-lib.rkt* 33⟩≡

```
#lang racket
⟨Dependencias de casos de prueba 4b⟩
(require "lambda-lib.rkt")
⟨Casos de prueba 4a⟩
```

# Capítulo 3

# Entorno interactivo

El sistema Lambda puede ser utilizado como biblioteca para otros programas que requieran trabajar con el cálculo $\lambda$, trabajando con Racket es suficiente importar el módulo `lambda-lib` o portando este módulo a otro lenguaje de programación y pasando las pruebas del archivo `pruebas-lambda-lib.rkt`.

En este capítulo se implementa un programa gráfico e interactivo para la exploración y estudio del cálculo $\lambda$ puro. Primero se presentan dos mecanismos para la edición de expresiones, un editor de texto y un editor estructural. Podiendo editar expresiones se aborda un mecanismo para la evaluación de expresiones desde la interfaz gráfica. La segunda parte de la interfaz consiste en un historial de expresiones evaluadas, en donde se pueden observar las entradas y salidas. Para hacer más amena la experiencia de escribir expresiones se presenta un visualizador con tres maneras de visualizar la expresión siendo editada: en código de Racket, con escritura formal y con escritura breve. Otra componente de la interfaz es un registro de metainstrucciones utilizado como referencia en la escritura de expresiones. Finalmente, todos estos elementos son combinados en una interfaz gráfica única concretando la experiencia completa de explorar el cálculo $\lambda$.

Ya que la programación de interfaces gráficas no es el enfoque de este trabajo, se reduce al mínimo la explicación del funcionamiento de esta parte de Lambda.

## 3.1 Editores

Para construir expresiones en Lambda se pueden utilizar dos mecanismos: el editor de texto y el editor estructural.

### 3.1.1 Editor de texto

35 ⟨*Editor de texto* 35⟩≡ (56)

```
(define-lex-abbrevs
  [paréntesis (:or #\( #\) #\[ #\])]
  [delimitadores (:or #\, #\. #\\ #\u3BB #\;)])

(define lexer-lambda
  (lexer
   [(:+ (:~ delimitadores paréntesis whitespace))
    (values lexeme 'otro #f
            (position-offset start-pos)
            (position-offset end-pos))]
```

```
    [(:+ whitespace)
     (values lexeme 'white-space #f
             (position-offset start-pos)
             (position-offset end-pos))]
    [delimitadores
     (values lexeme 'delimitadores #f
             (position-offset start-pos)
             (position-offset end-pos))]
    [paréntesis
     (values lexeme 'delimitadores (string->symbol lexeme)
             (position-offset start-pos)
             (position-offset end-pos))]
    [(eof)
     (values lexeme 'eof #f #f #f)]))

(define editor-texto%
  (class color:text%
    (super-new)
    (define expresión null)
    (define/augment (after-insert start len)
      (if (= len 1)
          (let ([cadena (send this get-text start (+ start 1))])
            (unless (char-whitespace? (string-ref cadena 0))
              (parsear)))
          (parsear)))
    (define/augment (after-delete start len)
      (parsear))
    (define/public (parsear)
      (with-handlers ([exn:fail? (lambda (err) (set! expresión null))])
        (let ([e (parsear-cadena (send this get-text))])
          (if (eof-object? e)
              (set! expresión null)
              (set! expresión e))))
      (reportar-cambio-editor expresión))
    (define/public (obtener-expresión)
      expresión)
    (define/public (cambiar-expresión e)
      (send this erase)
      (send this insert (if (null? e) "" (escribir-expresión-formal e)))
      (parsear))
    (send this start-colorer
          (lambda (what)
            (case what
              [(delimitadores paréntesis) "Realzado"]
              [else "Sencillo"]))
          lexer-lambda
          (list (list '|(| '|)|)
                (list '|[| '|]|)))
    (send this set-tabs null 2 #f)
    ;; Estilos
    (define estilo-realzado (make-object style-delta% 'change-size 10))
    (send estilo-realzado set-weight-on 'bold)
    (send estilo-realzado set-family 'modern)
    (editor:set-standard-style-list-delta "Realzado" estilo-realzado)
```

```
(define estilo-sencillo (make-object style-delta% 'change-size 10))
(send estilo-sencillo set-family 'modern)
(editor:set-standard-style-list-delta "Sencillo" estilo-sencillo)
;; Atajos
(define atajos (new keymap%))
(define-syntax atajo
  (syntax-rules ()
    [(_ (nombre . argumentos) teclas cuerpo ...)
     (let ([nombre* nombre]
           [teclas* teclas])
       (send atajos add-function nombre*
             (lambda argumentos
               cuerpo ...))
       (send atajos map-function teclas* nombre*))]))
(for-each (lambda (normal griego)
            (atajo ((format "inserta-~a" griego) editor evento)
                   (format "m:~a" normal)
                   (send editor insert (format "~a" griego))))
          (string->list "abcdefghijklmnoprstuvwxyzCDFGJLPSUVW")
          '(#\u3b1 #\u3b2 #\u3c8 #\u3b4 #\u3b5 #\u3c6 #\u3b3 #\u3b7 #\u3b9 #\u3be
                   #\u3ba #\u3bb #\u3bc #\u3bd #\u3bf #\u3c0 #\u3c1 #\u3c3 #\u3c4
                   #\u3b8 #\u3c9 #\u3c2 #\u3c7 #\u3c5 #\u3b6 #\u3a8 #\u394 #\u3a6
                   #\u393 #\u39e #\u39b #\u3a0 #\u3a3 #\u398 #\u3a9 #\u3a3))
(atajo ("selecciona-todo" editor evento) "c:a"
       (send editor do-edit-operation 'select-all))
(atajo ("deshacer" editor evento) "c:z"
       (send editor do-edit-operation 'undo))
(atajo ("rehacer" editor evento) "c:y"
       (send editor do-edit-operation 'redo))
(atajo ("copiar" editor evento) "c:c"
       (send editor do-edit-operation 'copy))
(atajo ("pegar" editor evento) "c:v"
       (send editor do-edit-operation 'paste))
(atajo ("cortar" editor evento) "c:x"
       (send editor do-edit-operation 'cut))
(atajo ("borrar-línea" editor evento) "c:k"
       (send editor do-edit-operation 'kill))
(atajo ("enviar-expresión" editor evento) "c:enter"
       (enviar-expresión (send editor obtener-expresión)))
(send this set-keymap atajos)))
```

Define:
editor-texto%, utilizado en fragmento 51.
Utiliza escribir-expresión-formal 24a, parsear 14a, y parsear-cadena 14b.

### 3.1.2 Editor estructural

37 ⟨*Editor estructural* 37⟩≡ (56)

```
(define fuente-texto-figura
  (make-object font% 12 'modern 'normal 'bold))

(define (text* x)
  (text x fuente-texto-figura))
```

```
(define (expresión->dibujo expresión cursor índice)
  (define (p-fondo p color)
    (let ([ancho (pict-width p)]
          [alto  (pict-height p)])
      (cc-superimpose (color (filled-rectangle ancho alto)) p)))
  (define (p-nombre e nombre)
    (if (and (eq? e cursor) (or (variable? e) (metainstrucción? e)))
        (let ([n (string-length nombre)])
          (if (< índice n)
              (p-fondo (hbl-append (text* (substring nombre 0 índice))
                                   (frame (blue (text* (substring nombre índice
                                                                  (+ índice 1))))
                                          #:color "blue"
                                          #:line-width .5)
                                   (text* (substring nombre (+ índice 1) n)))
                       red)
              (p-fondo (hbl-append (text* nombre)
                                   (frame (blue (text* "·")) #:color "blue")) red)))
        (let ([p (text* nombre)])
          (p-fondo p (if (eq? e cursor) red white)))))
  (define (p-lambda e argumentos)
    (define en-cursor? #f)
    (define figuras (map (lambda (x)
                           (if (and (abstracción? cursor)
                                    (eq? x (abstracción-argumento cursor)))
                               (begin
                                 (set! en-cursor? #t)
                                 (frame (naive-layered (escribir-dibujo-breve x))
                                        #:color "red" #:line-width 2))
                               (naive-layered (escribir-dibujo-breve x))))
                         argumentos))
    (p-fondo (apply hbl-append (p-nombre (if en-cursor? cursor e) (string #\u3bb))
                    (text* "  ")
                    (add-between figuras (text* "  ")))
             white))
  (define (e-variable e nombre)
    (tree-layout #:pict (p-nombre e nombre)))
  (define (e-abstracción e argumentos cuerpo)
    (tree-layout #:pict (p-lambda e argumentos)
                 (tree-edge #:edge-width 2
                            (escribir-dibujo-breve cuerpo))))
  (define (e-aplicación e operandos paréntesis)
    (define en-cursor? #f)
    (define hijos (map (lambda (x)
                         (if (and (aplicación? cursor)
                                  (eq? x (aplicación-operando cursor)))
                             (begin
                               (set! en-cursor? #t)
                               (escribir-dibujo-breve x))
                             (escribir-dibujo-breve x)))
                       operandos))
    (apply tree-layout #:pict (disk 10
                                    #:color (if en-cursor? "red" "white")
                                    #:border-width 2)
```

```
            (tree-edge #:edge-width 2
                       #:edge-color (if en-cursor? "red" "gray")
                       (car hijos))
            (map (lambda (e x)
                   (tree-edge #:edge-width 2
                              #:edge-color
                              (if en-cursor?
                                  (if (eq? e (aplicación-operando cursor))
                                      (begin
                                        (set! en-cursor? #f)
                                        "red")
                                      "red")
                                  "gray")
                              x))
                 (cdr operandos)
                 (cdr hijos))))
  (define (e-hueco e)
    (tree-layout #:pict (p-nombre e "[ ]")))
  (define (e-metainstrucción e nombre argumentos)
    (define en-cursor? (eq? e cursor))
    (tree-layout
     #:pict (p-nombre e nombre)
     (tree-edge #:edge-width 2
                #:edge-color (if en-cursor? "red" "gray")
                (tree-layout
                 #:pict
                 (p-fondo
                  (apply ht-append
                         (append (list ((if en-cursor? red black) (text* "[ ")))
                                 (add-between
                                  (map (lambda (x)
                                         (naive-layered (escribir-dibujo-breve x)))
                                       argumentos)
                                  ((if en-cursor? red black) (text* " , ")))
                                 (list ((if en-cursor? red black) (text* " ]")))))
                  white)))))
  (define escribir-dibujo-breve
    (escritor-breve e-variable e-abstracción e-aplicación
                    e-hueco e-metainstrucción))
  (if (null? expresión)
      (blank 50)
      (naive-layered (escribir-dibujo-breve expresión))))

(define (escalar-dibujo dibujo ancho alto)
  (let* ([dibujo-ancho (pict-width dibujo)]
         [dibujo-alto (pict-height dibujo)])
    (scale dibujo
           (min (* (/ ancho dibujo-ancho) .9)
                (* (/ alto dibujo-alto) .9)))))

(define editor-estructural%
  (class canvas%
    (inherit refresh-now get-dc get-width get-height)
    (super-new)
```

```
(define expresión null)
(define cursor expresión)
(define índice 0)
(define atajos (new keymap%))
(define/override (on-char evento)
  (let ([código (send evento get-key-code)])
    (if (and (char? código)
             (or (char-alphabetic? código)
                 (char-numeric? código)
                 (member código (string->list "+-_~:<>=?¿!@#$%^&*") char=?))
             (not (send evento get-control-down))
             (not (send evento get-meta-down)))
        (send this inserta (send evento get-key-code))
        (send atajos handle-key-event this evento))))
(define/override (on-paint)
  (define dc (get-dc))
  (define editor-ancho (get-width))
  (define editor-alto (get-height))
  (define dibujo (escalar-dibujo (expresión->dibujo expresión cursor índice)
                                 editor-ancho editor-alto))
  (define dibujo-ancho (pict-width dibujo))
  (define dibujo-alto (pict-height dibujo))
  (draw-pict dibujo dc
             (- (/ editor-ancho 2) (/ dibujo-ancho 2))
             (- (/ editor-alto 2) (/ dibujo-alto 2))))
(define/public (obtener-expresión)
  expresión)
(define/public (inserta caracter)
  (when (or (variable? cursor) (metainstrucción? cursor))
    (let ([selector (if (variable? cursor)
                        variable-nombre
                        metainstrucción-nombre)]
          [mutador (if (variable? cursor)
                       set-variable-nombre!
                       set-metainstrucción-nombre!)])
      (if (= índice (string-length (selector cursor)))
          (mutador cursor
                   (string-append
                    (selector cursor)
                    (string caracter)))
          (mutador cursor
                   (string-append
                    (substring (selector cursor) 0 índice)
                    (string caracter)
                    (substring (selector cursor)
                               índice
                               (string-length (selector cursor)))))))
    (set! índice (+ índice 1))
    (send this refresh-now)))
(define/public (elimina)
  (when (and (or (variable? cursor) (metainstrucción? cursor))
             (> índice 0)
             (> (string-length ((if (variable? cursor)
                                    variable-nombre
```

```
                                    metainstrucción-nombre)
                                  cursor))
                1))
    (let* ([selector (if (variable? cursor)
                         variable-nombre
                         metainstrucción-nombre)]
           [mutador (if (variable? cursor)
                        set-variable-nombre!
                        set-metainstrucción-nombre!)]
           [nombre (selector cursor)]
           [n (string-length nombre)])
      (if (= índice n)
          (mutador cursor
                   (substring nombre 0 (- índice 1)))
          (mutador cursor
                   (string-append (substring nombre 0 (- índice 1))
                                  (substring nombre índice n)))))
    (set! índice (- índice 1))
    (send this refresh-now)))
(define/public (borrar)
  (set! expresión null)
  (set! cursor expresión)
  (set! índice 0)
  (send this refresh-now))
(define/public (cambiar-expresión e)
  (set! expresión e)
  (set! cursor expresión)
  (set! índice 0)
  (send this refresh-now))
;; Atajos
(define-syntax atajo
  (syntax-rules ()
    [(_ (nombre . argumentos) teclas cuerpo ...)
     (let ([nombre* nombre]
           [teclas* teclas])
       (send atajos add-function nombre*
             (lambda argumentos
               cuerpo ...))
       (send atajos map-function teclas* nombre*))]))
(atajo ("expresión-abajo" editor evento) "c:s"
       (cond [(abstracción? cursor)
              (set! cursor (abstracción-cuerpo cursor))
              (set! índice 0)
              (send editor refresh-now)]
             [(aplicación? cursor)
              (set! cursor (aplicación-operando cursor))
              (set! índice 0)
              (send editor refresh-now)]
             [(and (metainstrucción? cursor)
                   (not (null? (metainstrucción-argumentos cursor))))
              (set! cursor (car (metainstrucción-argumentos cursor)))
              (set! índice 0)
              (send editor refresh-now)]))
(atajo ("expresión-izquierda" editor evento) "c:a"
```

```
      (cond [(aplicación? cursor)
             (set! cursor (aplicación-operador cursor))
             (set! índice 0)
             (send editor refresh-now)]))
(atajo ("expresión-derecha" editor evento) "c:d"
      (cond [(abstracción? cursor)
             (set! cursor (abstracción-argumento cursor))
             (set! índice 0)
             (send editor refresh-now)]))
(letrec ([recorre
          (lambda (x)
            (cond [(variable? x) #f]
                  [(abstracción? x)
                   (if (or (eq? cursor (abstracción-cuerpo x))
                           (eq? cursor (abstracción-argumento x)))
                       x
                       (recorre (abstracción-cuerpo x)))]
                  [(aplicación? x)
                   (if (or (eq? cursor (aplicación-operador x))
                           (eq? cursor (aplicación-operando x)))
                       x
                       (cond [(recorre (aplicación-operador x))
                              => identity]
                             [else (recorre (aplicación-operando x))]))]
                  [(hueco? x) #f]
                  [(metainstrucción? x)
                   (if (memq cursor (metainstrucción-argumentos x))
                       x
                       (let itera ([argumentos (metainstrucción-argumentos x)])
                         (cond [(null? argumentos) #f]
                               [(recorre (car argumentos))
                                => identity]
                               [else (itera (cdr argumentos))])))]
                  [else #f]))])
  (atajo ("expresión-arriba" editor evento) "c:w"
        (cond [(recorre expresión)
               => (lambda (x)
                    (set! cursor x)
                    (set! índice 0)
                    (send editor refresh-now))]))
  (atajo ("expresión-siguiente" editor evento) "c:e"
        (cond [(recorre expresión)
               => (lambda (p)
                    (when (metainstrucción? p)
                      (let itera ([argumentos (metainstrucción-argumentos p)])
                        (cond [(null? argumentos) (void)]
                              [(null? (cdr argumentos)) (void)]
                              [(eq? (car argumentos) cursor)
                               (set! cursor (cadr argumentos))
                               (set! índice 0)
                               (send editor refresh-now)]
                              [else (itera (cdr argumentos))]))))]))
  (atajo ("expresión-anterior" editor evento) "c:q"
        (cond [(recorre expresión)
```

```
              => (lambda (p)
                   (when (metainstrucción? p)
                     (let itera ([argumentos (metainstrucción-argumentos p)])
                       (cond [(null? argumentos) (void)]
                             [(null? (cdr argumentos)) (void)]
                             [(eq? (cadr argumentos) cursor)
                              (set! cursor (car argumentos))
                              (set! índice 0)
                              (send editor refresh-now)]
                             [else (itera (cdr argumentos))])))))]))
(atajo ("variable-aquí" editor evento) "c:1"
       (let ([x (variable "x")])
         (cond [(recorre expresión)
                => (lambda (p)
                     (cond [(abstracción? p)
                            (if (eq? cursor (abstracción-argumento p))
                                (set-abstracción-argumento! p x)
                                (set-abstracción-cuerpo! p x))]
                           [(aplicación? p)
                            (if (eq? cursor (aplicación-operador p))
                                (set-aplicación-operador! p x)
                                (set-aplicación-operando! p x))]
                           [(metainstrucción? p)
                            (set-metainstrucción-argumentos! p
                              (map (lambda (e)
                                     (if (eq? e cursor) x e))
                                   (metainstrucción-argumentos p)))]))]
               [else
                (set! expresión x)])
         (set! cursor x)
         (set! índice 0)
         (send editor refresh-now)))
(atajo ("abstracción-aquí" editor evento) "c:2"
       (let ([x (abstracción (variable "x") (variable "x"))])
         (cond [(recorre expresión)
                => (lambda (p)
                     (cond [(abstracción? p)
                            (set-abstracción-cuerpo! p x)]
                           [(aplicación? p)
                            (if (eq? cursor (aplicación-operador p))
                                (set-aplicación-operador! p x)
                                (set-aplicación-operando! p x))]
                           [(metainstrucción? p)
                            (set-metainstrucción-argumentos! p
                              (map (lambda (e)
                                     (if (eq? e cursor) x e))
                                   (metainstrucción-argumentos p)))]))]
               [else
                (set! expresión x)])
         (set! cursor x)
         (set! índice 0)
         (send editor refresh-now)))
(atajo ("aplicación-aquí" editor evento) "c:3"
       (let ([x (aplicación (variable "x") (variable "x"))])
```

```
          (cond [(recorre expresión)
                 => (lambda (p)
                      (cond [(abstracción? p)
                             (set-abstracción-cuerpo! p x)]
                            [(aplicación? p)
                             (if (eq? cursor (aplicación-operador p))
                                 (set-aplicación-operador! p x)
                                 (set-aplicación-operando! p x))]
                            [(metainstrucción? p)
                             (set-metainstrucción-argumentos! p
                              (map (lambda (e)
                                     (if (eq? e cursor) x e))
                                   (metainstrucción-argumentos p)))]))]
                [else
                 (set! expresión x)])
          (set! cursor x)
          (set! índice 0)
          (send editor refresh-now)))
  (atajo ("hueco-aquí" editor evento) "c:4"
         (let ([x (hueco)])
           (cond [(recorre expresión)
                  => (lambda (p)
                       (cond [(abstracción? p)
                              (set-abstracción-cuerpo! p x)]
                             [(aplicación? p)
                              (if (eq? cursor (aplicación-operador p))
                                  (set-aplicación-operador! p x)
                                  (set-aplicación-operando! p x))]
                             [(metainstrucción? p)
                              (set-metainstrucción-argumentos! p
                               (map (lambda (e)
                                      (if (eq? e cursor) x e))
                                    (metainstrucción-argumentos p)))]))]
                 [else
                  (set! expresión x)])
           (set! cursor x)
           (set! índice 0)
           (send editor refresh-now)))
  (let ([metainstrucción-aquí-n
         (lambda (n)
           (lambda (editor evento)
             (let ([x (metainstrucción "f" (build-list n (lambda (i)
                                                           (variable "x"))))])
               (cond [(recorre expresión)
                      => (lambda (p)
                           (cond [(abstracción? p)
                                  (set-abstracción-cuerpo! p x)]
                                 [(aplicación? p)
                                  (if (eq? cursor (aplicación-operador p))
                                      (set-aplicación-operador! p x)
                                      (set-aplicación-operando! p x))]
                                 [(metainstrucción? p) p
                                  (set-metainstrucción-argumentos! p
                                   (map (lambda (e)
```

```
                                                (if (eq? e cursor) x e))
                                              (metainstrucción-argumentos p)))]))]
                                [else
                                 (set! expresión x)])
                          (set! cursor x)
                          (set! índice 0)
                          (send editor refresh-now))))])
              (for-each (lambda (i)
                          (send atajos add-function (format "metainstrucción-aquí-~a" i)
                                (metainstrucción-aquí-n i))
                          (send atajos map-function
                                (format "c:5;~a" i)
                                (format "metainstrucción-aquí-~a" i)))
                        (range 0 10))))
      (atajo ("índice-derecha" editor evento) "right"
             (cond [(variable? cursor)
                    (when (< índice (string-length (variable-nombre cursor)))
                      (set! índice (+ índice 1))
                      (send editor refresh-now))]
                   [(metainstrucción? cursor)
                    (when (< índice (string-length (metainstrucción-nombre cursor)))
                      (set! índice (+ índice 1))
                      (send editor refresh-now))]))
      (atajo ("índice-izquierda" editor evento) "left"
             (cond [(or (variable? cursor) (metainstrucción? cursor))
                    (when (> índice 0)
                      (set! índice (- índice 1))
                      (send editor refresh-now))]))
      (for-each (lambda (normal griego)
                  (atajo ((format "inserta-~a" griego) editor evento)
                         (format "m:~a" normal)
                         (send editor inserta griego)))
                (string->list "abcdefghijkmnoprstuvwxyzCDFGJLPSUVW")
                '(#\u03b1 #\u03b2 #\u03c8 #\u03b4 #\u03b5 #\u03c6 #\u03b3 #\u03b7 #\u03b9 #\u03be
                          #\u03ba  #\u03bc #\u03bd #\u03bf #\u03c0 #\u03c1 #\u03c3 #\u03c4
                          #\u03b8 #\u03c9 #\u03c2 #\u03c7 #\u03c5 #\u03b6 #\u03a8 #\u394 #\u03a6
                          #\u393 #\u39e #\u39b #\u03a0 #\u03a3 #\u398 #\u03a9 #\u03a3))
      (atajo ("eliminar-texto" editor evento) "backspace"
             (send editor elimina))
      (atajo ("enviar-expresión" editor evento) "c:enter"
             (enviar-expresión (send editor obtener-expresión)))))
```

Define:
editor-estructural%, utilizado en fragmento 51.
escalar-dibujo, utilizado en fragmento 47.
Utiliza abstracción 4c, aplicación 5a, escritor-breve 27, hueco 5c, metainstrucción 6a, y variable 3b.

## 3.2 Historial

45 ⟨*Historial* 45⟩≡ (56)

```
(define expresión-snip%
  (class editor-snip%
    (init-field formal
                breve)
```

```
      (super-new)
      (define actual breve)
      (define editor (new text%))
      (send this set-editor editor)
      (refrescar)
      (define/override (on-event dc x y ex ey evento)
        (case (send evento get-event-type)
          [(left-down)
           (enviar-texto actual)
           (enfocar-editor)]
          [(right-down)
           (if (eq? actual breve)
               (set! actual formal)
               (set! actual breve))
           (refrescar)
           (enfocar-editor)]
          [else
           (enfocar-editor)]))
      (define/public (refrescar)
        (send editor erase)
        (send editor insert actual))))

  (define evaluación%
    (class object%
      (init-field entrada)
      (super-new)
      (define entrada-formal (escribir-expresión-formal entrada))
      (define entrada-breve (escribir-expresión-breve entrada))
      (define salida
        (with-handlers ([exn:fail? (lambda (err)
                                     (cons 'error (exn-message err)))])
          (evaluar-expresión entrada)))
      (define salida-formal
        (with-handlers ([exn:fail? (lambda (err)
                                     (format "~v" salida))])
          (if (and (pair? salida) (eq? 'error (car salida)))
              (cdr salida)
              (escribir-expresión-formal salida))))
      (define salida-breve
        (with-handlers ([exn:fail? (lambda (err)
                                     (format "~v" salida))])
          (if (and (pair? salida) (eq? 'error (car salida)))
              (cdr salida)
              (escribir-expresión-breve salida))))
      (define/public (obtener-entrada)
        entrada)
      (define/public (obtener-entrada-formal)
        entrada-formal)
      (define/public (obtener-entrada-breve)
        entrada-breve)
      (define/public (obtener-entrada-snip)
        (new expresión-snip% [formal entrada-formal] [breve entrada-breve]))
      (define/public (obtener-salida)
        salida)
```

```
    (define/public (obtener-salida-formal)
      salida-formal)
    (define/public (obtener-salida-breve)
      salida-breve)
    (define/public (obtener-salida-snip)
      (new expresión-snip% [formal salida-formal] [breve salida-breve]))))


(define historial%
  (class text%
    (super-new)
    (define evaluaciones null)
    (define bloqueado? #t)
    (define/augment (can-insert? s l) (not bloqueado?))
    (define/augment (can-delete? s l) #f)
    (define/public (insertar evaluación)
      (define (al-final x)
        (send this insert x (send this last-position)))
      (set! evaluaciones (cons evaluación evaluaciones))
      (set! bloqueado? #f)
      (al-final #\newline)
      (al-final (send evaluación obtener-entrada-snip))
      (al-final (new pict-snip% [pict (arrow 10 0)]))
      (al-final (send evaluación obtener-salida-snip))
      (al-final #\newline)
      (set! bloqueado? #t))
    (send this hide-caret #t)))
```

Define:
evaluación%, utilizado en fragmento 51.
expresión-snip%, nunca utilizado.
historial%, utilizado en fragmento 51.
Utiliza escribir-expresión-breve 29, escribir-expresión-formal 24a, y evaluar-expresión 31a.

## 3.3 Visualizador

47 ⟨*Visualizador* 47⟩≡ (56)

```
(define (expresión->código expresión [columnas 50])
  (codeblock-pict
   #:keep-lang-line? #f
   (string-append "#lang racket\n"
                  (pretty-format expresión columnas))))

(define (expresión->figura-escrita-formal expresión)
  (define (e-variable e nombre)
    (colorize (text nombre) (dark "green")))
  (define (e-abstracción e argumento cuerpo)
    (hbl-append (colorize (text (string-append "(" (string #\u3BB))) (dark "brown"))
                (escribir-dibujo-formal argumento)
                (colorize (text ".") (dark "brown"))
                (escribir-dibujo-formal cuerpo)
                (colorize (text ")") (dark "brown"))))
  (define (e-aplicación e operador operando)
    (hbl-append (colorize (text "(") (dark "brown"))
```

```
                (escribir-dibujo-formal operador)
                (text " ")
                (escribir-dibujo-formal operando)
                (colorize (text ")") (dark "brown"))))
  (define (e-hueco e)
    (colorize (text "[ ]") (dark "brown")))
  (define (e-metainstrucción e nombre argumentos)
    (apply hbl-append
           (colorize (text nombre) (dark "green"))
           (colorize (text "[") (dark "brown"))
           (append (add-between (map escribir-dibujo-formal argumentos)
                                (colorize (text ",") (dark "brown")))
                   (list (colorize (text "]") (dark "brown"))))))
  (define escribir-dibujo-formal
    (escritor-formal e-variable e-abstracción e-aplicación
                     e-hueco e-metainstrucción))
  (if (null? expresión)
      (blank 50)
      (escribir-dibujo-formal expresión)))

(define (expresión->figura-escrita-breve expresión)
  (define (e-variable e nombre)
    (colorize (text nombre) (dark "green")))
  (define (e-abstracción e argumentos cuerpo)
    (apply hbl-append
           (colorize (text (string #\u3BB)) (dark "brown"))
           (append (add-between (map escribir-dibujo-breve argumentos)
                                (text " "))
                   (list (colorize (text ".") (dark "brown"))
                         (escribir-dibujo-breve cuerpo)))))
  (define (e-aplicación e operandos paréntesis)
    (apply hbl-append
           (add-between (map (lambda (x p?)
                               (if p?
                                   (hbl-append (colorize (text "(") (dark "brown"))
                                               (escribir-dibujo-breve x)
                                               (colorize (text ")") (dark "brown")))
                                   (escribir-dibujo-breve x)))
                             operandos
                             paréntesis)
                        (text " "))))
  (define (e-hueco e)
    (colorize (text "[ ]") (dark "brown")))
  (define (e-metainstrucción e nombre argumentos)
    (apply hbl-append
           (colorize (text nombre) (dark "green"))
           (colorize (text "[") (dark "brown"))
           (append (add-between (map escribir-dibujo-breve argumentos)
                                (colorize (text ",") (dark "brown")))
                   (list (colorize (text "]") (dark "brown"))))))
  (define escribir-dibujo-breve
    (escritor-breve e-variable e-abstracción e-aplicación
                    e-hueco e-metainstrucción))
  (if (null? expresión)
```

```
      (blank 50)
      (escribir-dibujo-breve expresión)))

(define código-lienzo%
  (class canvas%
    (inherit refresh refresh-now get-dc get-width get-height)
    (super-new)
    (send this init-auto-scrollbars 1000 1000 0.0 0.0)
    (define expresión null)
    (define expresión-pict (blank 50))
    (define/override (on-paint)
      (define dc (get-dc))
      (draw-pict expresión-pict dc 0 0))
    (define/public (actualizar-expresión e)
      (set! expresión e)
      (set! expresión-pict
            (scale (expresión->código e (inexact->exact
                                         (floor (/ (get-width) 15))))
                   2))
      (send this refresh-now))))

(define figura-escrita-formal-lienzo%
  (class canvas%
    (inherit refresh refresh-now get-dc get-width get-height)
    (super-new)
    (define expresión null)
    (define expresión-pict (expresión->figura-escrita-formal expresión))
    (define/override (on-paint)
      (define dc (get-dc))
      (define lienzo-ancho (get-width))
      (define lienzo-alto (get-height))
      (define dibujo (escalar-dibujo expresión-pict
                                     lienzo-ancho lienzo-alto))
      (define dibujo-ancho (pict-width dibujo))
      (define dibujo-alto (pict-height dibujo))
      (draw-pict dibujo dc
                 (- (/ lienzo-ancho 2) (/ dibujo-ancho 2))
                 (- (/ lienzo-alto 2) (/ dibujo-alto 2))))
    (define/public (actualizar-expresión e)
      (set! expresión e)
      (set! expresión-pict
            (expresión->figura-escrita-formal e))
      (send this refresh-now))))

(define figura-escrita-breve-lienzo%
  (class canvas%
    (inherit refresh refresh-now get-dc get-width get-height)
    (super-new)
    (define expresión null)
    (define expresión-pict (expresión->figura-escrita-breve expresión))
    (define/override (on-paint)
      (define dc (get-dc))
      (define lienzo-ancho (get-width))
      (define lienzo-alto (get-height))
```

```
        (define dibujo (escalar-dibujo expresión-pict
                                       lienzo-ancho lienzo-alto))
        (define dibujo-ancho (pict-width dibujo))
        (define dibujo-alto (pict-height dibujo))
        (draw-pict dibujo dc
                   (- (/ lienzo-ancho 2) (/ dibujo-ancho 2))
                   (- (/ lienzo-alto 2) (/ dibujo-alto 2))))
      (define/public (actualizar-expresión e)
        (set! expresión e)
        (set! expresión-pict (expresión->figura-escrita-breve e))
        (send this refresh-now))))
```

Define:
código-lienzo%, utilizado en fragmento 51.
figura-escrita-breve-lienzo%, utilizado en fragmento 51.
figura-escrita-formal-lienzo%, utilizado en fragmento 51.
Utiliza abstracción 4c, aplicación 5a, escalar-dibujo 37, escritor-breve 27, escritor-formal 23a, hueco 5c, metainstrucción 6a, y variable 3b.

## 3.4 Registro de metainstrucciones

50 ⟨*Registro de metainstrucciones* 50⟩≡ (56)

```
(pretty-print-current-style-table
 (pretty-print-extend-style-table (pretty-print-current-style-table)
                                  '(define-metainstrucción match)
                                  '(define case)))

(pretty-print-columns 80)

(define metainstrucciones-código (make-hash))

(define-syntax define-metainstrucción
  (syntax-rules ()
    [(define-metainstrucción (id . args) cuerpo ...)
     (begin (define (id . args) cuerpo ...)
            (hash-set! metainstrucciones
                       (symbol->string 'id)
                       id)
            (hash-set! metainstrucciones-código
                       (symbol->string 'id)
                       '(define-metainstrucción (id . args) cuerpo ...)))]))

(define (cargar-archivo-metainstrucciones archivo)
  (let ([file (open-input-file archivo)])
    (let loop ([e (read file)])
      (unless (eof-object? e)
        (eval e)
        (loop (read file)))))
  (reportar-cambio-registro))

(define código-racket%
  (class racket:text%
    (super-new)
    (define bloqueado? #t)
```

```
(define/augment (can-insert? s l) (not bloqueado?))
(define/augment (can-delete? s l) #t)
(define/public (insertar sexp)
  (set! bloqueado? #f)
  (send this erase)
  (let ([p (open-output-string)])
    (pretty-print sexp p 1)
    (let ([cadena (get-output-string p)])
      (for-each (lambda (x)
                  (send this insert x)
                  (send this insert-return))
                (map string-trim (string-split cadena "\n"))))
  (set! bloqueado? #t)))))
```

Utiliza define-metainstrucción 6c, metainstrucciones 6c, y metainstrucción 6a.

## 3.5 Interfaz gráfica

51 ⟨*Uniendo las componentes de la interfaz gráfica* 51⟩≡ (56)

```
;; Ventana registro

(define ventana-registro
  (new frame%
       [label "Registro Metainstrucciones"]
       [width 800]
       [height 600]))

(define registro-panel
  (new panel:horizontal-dragable%
       [parent ventana-registro]
       [border 0]
       [spacing 20]))

(define listado-panel
  (new vertical-panel%
       [parent registro-panel]
       [spacing 20]))

(define listado-metainstrucciones
  (new list-box%
       [parent listado-panel]
       [label #f]
       [choices (list "")]
       [style (list 'single
                    'column-headers)]
       [columns (list "Nombre")]
       [callback
        (lambda (listado evento)
          (when (eq? 'list-box (send evento get-event-type))
            (let [(seleccionados (send listado get-selections))]
              (unless (null? seleccionados)
                (send código-texto insertar
                      (send listado get-data (car seleccionados)))))))]))
```

```
(define botón-cargar
  (new button%
       [label "Cargar archivo"]
       [parent listado-panel]
       [stretchable-width #t]
       [callback
        (lambda (botón evento)
          (when (eq? 'button (send evento get-event-type))
            (let ([p (get-file "Cargar archivo de metainstrucciones")])
              (when p
                (cargar-archivo-metainstrucciones (path->string p))))))]))

(define código-texto (new código-racket%))

(define lienzo-implementación (new canvas:color%
                                   [parent registro-panel]
                                   [editor código-texto]))

(define ventana-visualizador
  (new frame%
       [label "Visualizador"]
       [width 800]  [min-width 200]
       [height 600] [min-height 200]))

(define visualizador-panel
  (new vertical-panel%
       [parent ventana-visualizador]
       [border 0]
       [spacing 20]))

(define visualizador-código
  (new código-lienzo%
       [parent visualizador-panel]
       [style (list 'vscroll 'hscroll)]))

(define visualizador-formal
  (new figura-escrita-formal-lienzo%
       [parent visualizador-panel]
       [stretchable-height #f]
       [min-height 50]))

(define visualizador-breve
  (new figura-escrita-breve-lienzo%
       [parent visualizador-panel]
       [stretchable-height #f]
       [min-height 50]))

(define ventana-principal
  (new frame%
       [label "Lambda"]
       [width 800]  [min-width 200]
       [height 600] [min-height 200]))

(define panel-principal
```

```
  (new vertical-panel%
       [parent ventana-principal]))

(define panel-herramientas
  (new horizontal-panel%
       [parent panel-principal]
       [border 0]
       [stretchable-height #f]))

(define botón-visualizador
  (new button%
       [parent panel-herramientas]
       [label "visualizador"]
       [callback
        (lambda (botón evento)
          (send ventana-visualizador show #t))]))

(define botón-metainstrucciones
  (new button%
       [parent panel-herramientas]
       [label "metainstrucciones"]
       [callback
        (lambda (botón evento)
          (send ventana-registro show #t))]))

(define herramientas-espacio
  (new panel%
       [parent panel-herramientas]))

(define botón-ayuda
  (new button%
       [parent panel-herramientas]
       [label "ayuda"]))

(define panel-historial/editor
  (new panel:vertical-dragable%
       [parent panel-principal]))

(define historial-panel
  (new panel%
       [parent panel-historial/editor]
       [border 0]))

(define historial (new historial%))

(define historial-lienzo
  (new editor-canvas%
       [parent historial-panel]
       [editor historial]))

(define panel-tabs
  (new tab-panel%
       [parent panel-historial/editor]
       [choices (list "Texto" "Estructura")]
```

```
        [callback
         (lambda (panel evento)
           (case (send panel get-selection)
             [(0)
              (send panel change-children
                    (lambda (x)
                      (list editor-texto-panel)))
              (reportar-otro-editor 'Texto)]
             [(1)
              (send panel change-children
                    (lambda (x)
                      (list editor-estructural-panel)))
              (reportar-otro-editor 'Estructura)]))]))

(define editor-texto-panel
  (new panel%
       [parent panel-tabs]
       [border 0]))

(define editor-texto (new editor-texto%))

(define editor-texto-lienzo
  (new editor-canvas%
       [parent editor-texto-panel]
       [editor editor-texto]))

(define editor-estructural-panel
  (new panel%
       [parent panel-tabs]
       [border 0]))

(define editor-estructural
  (new editor-estructural%
       [parent editor-estructural-panel]))

(send panel-tabs change-children
      (lambda (x)
        (list editor-texto-panel)))

;;
;; Procedimientos conectivos
;;

(define (enviar-expresión expresión)
  (send historial insertar
        (new evaluación% [entrada expresión]))
  (case editor-actual
    [(Texto)
     (send editor-texto erase)]
    [(Estructura)
     (send editor-estructural borrar)]))

(define (reportar-cambio-editor expresión)
  (send visualizador-código actualizar-expresión expresión)
```

```
  (send visualizador-formal actualizar-expresión expresión)
  (send visualizador-breve actualizar-expresión expresión))

(define (enfocar-editor)
  (case editor-actual
    [(Texto)
     (send editor-texto-lienzo focus)]
    [(Estructura)
     (send editor-estructural focus)]))

(define (enviar-texto cadena)
  (case editor-actual
    [(Texto)
     (send editor-texto erase)
     (send editor-texto insert cadena)]
    [(Estructura)
     (send editor-estructural borrar)
     (send editor-estructural cambiar-expresión
           (parsear-cadena cadena))]))

(define editor-actual 'Texto)

(define (reportar-otro-editor objetivo)
  (unless (eq? editor-actual objetivo)
    (case editor-actual
      [(Texto)
       (send editor-estructural cambiar-expresión
             (send editor-texto obtener-expresión))]
      [(Estructura)
       (send editor-texto cambiar-expresión
             (send editor-estructural obtener-expresión))]))
  (set! editor-actual objetivo)
  (enfocar-editor))

(define (reportar-cambio-registro)
  (hash-for-each metainstrucciones
                 (lambda (nombre proc)
                   (send listado-metainstrucciones append
                         nombre (hash-ref metainstrucciones-código nombre)))
                 #t))

;;
;; Mostrar la ventana principal
;;

(send ventana-principal show #t)
```

Utiliza código-lienzo% 47, editor-estructural% 37, editor-texto% 35, evaluación% 45, figura-escrita-breve-lienzo% 47, figura-escrita-formal-lienzo% 47, historial% 45, metainstrucciones 6c, parsear 14a, y parsear-cadena 14b.

## 3.6 Código fuente

56 ⟨*lambda-gui.rkt* 56⟩≡

```
#lang racket/gui
(require "lambda-lib.rkt")
(require framework)
(require (except-in parser-tools/lex
                    blank))
(require (prefix-in : parser-tools/lex-sre))
(require pict)
(require pict/snip)
(require pict/color)
(require pict/code)
(require pict/tree-layout)

⟨Editor de texto 35⟩
⟨Editor estructural 37⟩
⟨Historial 45⟩
⟨Visualizador 47⟩
⟨Registro de metainstrucciones 50⟩
⟨Uniendo las componentes de la interfaz gráfica 51⟩
```

# Capítulo 4

# Extensiones

En este capítulo se muestran ejemplos de metainstrucciones, estos procedimientos podrán ser incluídos por programas que usen `lambda-lib` como por ejemplo `lambda-gui`. Las metainstrucciones presentes se dividen en tres categorías: análisis de estructura de términos, transformación de términos y manejo de codificaciones.

## 4.1 Análisis de términos

57a ⟨*Metainstrucciones* 57a⟩≡ (60b) 57b▸

```
(define-metainstrucción (longitud M)
  (if (término? M)
      (let recurre ([M M])
        (match M
          [(variable x)
           1]
          [(abstracción x M)
           (+ 1 (recurre M))]
          [(aplicación M N)
           (+ (recurre M) (recurre N))]))
      -1))
```

Define:
`longitud`, utilizado en fragmento 20b.
Utiliza `abstracción` 4c, `aplicación` 5a, `define-metainstrucción` 6c, `metainstrucción` 6a, `término?` 7c, y `variable` 3b.

57b ⟨*Metainstrucciones* 57a⟩+≡ (60b) ◂57a 58a▸

```
(define-metainstrucción (variables-libres M)
  (match M
    [(variable x)
     (list (variable x))]
    [(abstracción x M)
     (remove x (variables-libres M))]
    [(aplicación M N)
     (remove-duplicates
      (append (variables-libres M)
              (variables-libres N)))]
    [_ null]))
```

Define:
`variables-libres`, utilizado en fragmento 58.

Utiliza abstracción 4c, aplicación 5a, define-metainstrucción 6c, metainstrucción 6a, y variable 3b.

## 4.2 Transformación de términos

58a ⟨*Metainstrucciones* 57a⟩+≡ (60b) ◂57b 58b▸

```
(define-metainstrucción (sustituye M x N)
  (match M
    [(== x)
     N]
    [(variable y)
     M]
    [(aplicación P Q)
     (aplicación (sustituye P x N) (sustituye Q x N))]
    [(abstracción (== x) P)
     M]
    [(abstracción y P)
     (cond [(not (member x (variables-libres P)))
            (abstracción y P)]
           [(not (member y (variables-libres N)))
            (abstracción y (sustituye P x N))]
           [else
            (let ([z (variable-ajena (aplicación N P))])
              (abstracción z (sustituye (sustituye P y z) x N)))])]
    [_ M]))
```

Define:
sustituye, nunca utilizado.
Utiliza abstracción 4c, aplicación 5a, define-metainstrucción 6c, metainstrucción 6a, variable 3b, variable-ajena 58b, y variables-libres 57b.

58b ⟨*Metainstrucciones* 57a⟩+≡ (60b) ◂58a 59a▸

```
(define-metainstrucción (variable-ajena M)
  (let ([variables (variables-libres M)])
    (let itera ([letra #\a])
      (if (char>? letra #\z)
          (let itera [(letra #\a) (número 2)]
            (if (char>? letra #\z)
                (itera #\a (+ número 1))
                (let ([v (variable (string-append (string letra)
                                                  (number->string número)))])
                  (if (member v variables)
                      (itera (integer->char (+ (char->integer letra) 1))
                             número)
                      v))))
          (let ([v (variable (string letra))])
            (if (member v variables)
                (itera (integer->char (+ (char->integer letra) 1)))
                v))))))
```

Define:
variable-ajena, utilizado en fragmento 58a.
Utiliza define-metainstrucción 6c, metainstrucción 6a, variable 3b, y variables-libres 57b.

59a ⟨*Metainstrucciones* 57a⟩+≡ (60b) ◂58b 59b▸
```
(define-metainstrucción (llenar-huecos M N)
  (match M
    [(variable x)
     M]
    [(abstracción x P)
     (abstracción x (llenar-huecos P N))]
    [(aplicación P Q)
     (aplicación (llenar-huecos P N)
                 (llenar-huecos Q N))]
    [(hueco)
     N]
    [(metainstrucción x Ps)
     (metainstrucción x (map (lambda (P)
                               (llenar-huecos P N)) Ps))]
    [_ M]))
```
Define:
llenar-huecos, nunca utilizado.
Utiliza abstracción 4c, aplicación 5a, define-metainstrucción 6c, hueco 5c, metainstrucción 6a, y variable 3b.

## 4.3 Codificaciones

59b ⟨*Metainstrucciones* 57a⟩+≡ (60b) ◂59a 59c▸
```
(define-metainstrucción (T)
  (parsear-cadena "\\x y.x"))

(define-metainstrucción (F)
  (parsear-cadena "\\x y.y"))
```
Define:
F, nunca utilizado.
T, nunca utilizado.
Utiliza define-metainstrucción 6c, metainstrucción 6a, parsear 14a, y parsear-cadena 14b.

59c ⟨*Metainstrucciones* 57a⟩+≡ (60b) ◂59b 60a▸
```
(define-metainstrucción (decodifica-numeral M)
  (match M
    [(abstracción (variable f)
                  (abstracción (variable x) M))
     (let itera ([M M] [n 0])
       (match M
         [(== (variable x)) n]
         [(aplicación (== (variable f)) M)
          (itera M (+ n 1))]
         [_ -1]))]
    [_ -1]))
```
Define:
decodifica-numeral, nunca utilizado.
Utiliza abstracción 4c, aplicación 5a, define-metainstrucción 6c, metainstrucción 6a, y variable 3b.

60a ⟨*Metainstrucciones* 57a⟩+≡ (60b) ◃59c

```
(define-metainstrucción (codifica-número M)
  (match M
    [(variable x)
     (let ([n (string->number x)])
       (if (and (number? n) (integer? n) (>= n 0))
           (abstracción (variable "f")
                        (abstracción (variable "x")
                                     (let itera ([n n])
                                       (if (zero? n)
                                           (variable "x")
                                           (aplicación (variable "f")
                                                       (itera (- n 1)))))))
           M))]
    [_ M]))
```

Define:
codifica-número, nunca utilizado.
Utiliza abstracción 4c, aplicación 5a, define-metainstrucción 6c, metainstrucción 6a, y variable 3b.

60b ⟨*lambda-init.rkt* 60b⟩≡

```
⟨Metainstrucciones 57a⟩
```