Lambda

Un sistema computacional basado en el cálculo λ

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

INTRODUCCIÓN

El cálculo λ 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 el cálculo, es necesario entender su formalización.

Una alternativa a esta manera de estudiar el cálculo λ es escribiendo un programa de computadora con el que se pueda trabajar de manera interactiva. Leyendo el código presente es posible entender de manera precisa los objetos matemáticos y procedimientos computacionales asociados al cálculo, de tal manera que extensiones y modificaciones sean fáciles de incorporar.

Para poder materializar este método de aprendizaje del cálculo λ se identifican tres etapas: primero está la implementación de las estructuras y algoritmos que permiten construír al cálculo $\lambda K\beta$; después está la implementación de una interfaz con la que se puede explorar y extender el programa base; finalmente está la implementación de las extensiones del programa base.

El código de este trabajo está escrito en el lenguaje de programación `Racket`, un dialecto de `Lisp` similar a `Scheme`. Traducir este programa a otros lenguajes de programación puede ser un buen ejercicio para comprender mejor la idea central del código.

CAPÍTULO I

PIEZAS FUNDAMENTALES

I. Objetos elementales

Existen diversas clases de objetos que pueden existir en el mundo del cálculo λ, estas clases de objetos representan ideas abstractas asociadas a cadenas de un lenguaje formal denotado Λ. 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 Λ son llamados *términos*, en este programa se implementan los tres tipos de términos más básicos del cálculo λ: las *variables*, las *abstracciones* y las *aplicaciones*. Las clases de objetos del sistema `Lambda` tienen como representación simbólica elementos del lenguaje formal Ω llamados *expresiones*, los elementos de Ω 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 Ω.

1 ⟨*Definición de expresión* 1⟩≡
```
(define/contract (expresión? x)
  (any/c . -> . boolean?)
  (or (variable? x) (abstracción? x) (aplicación? x)
      (hueco? x) (metainstrucción? x)))
```
Define:
`expresión?`, usado en fragmentos 1, 5, 1, 7, 1, 11, 18, 32–34, 48, 56, 61, 66, 78–80, y 85.
Hace referencia a `abstracción` 5, `aplicación` 7, `hueco` 9, `metainstrucción` 11, y `variable` 2.
Este código es usado en fragmento 27.

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?`.

2 ⟨*Definición de variable* 2⟩≡
```
(define-struct/contract variable
  ([nombre string?])
  #:transparent
  #:mutable)
```

Define:
variable, usado en fragmentos 1–3, 5, 6, 8, 12, 14, 16, 17, 2, 19, 20, 22, 24, 26, 28, 37, 42, 45, 49, 56, 60, 62, 63, 65–67, 77–79, 81, 87, 91, 93, 97–101, 103, y 104.
Este código es usado en fragmento 27.

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".

3 ⟨*Casos de prueba* 3⟩≡
```
(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"))
```
Hace referencia a variable 2.
Esta definición se continua en fragmentos 6, 8, 10, 12, 14, 17, 20, 22, 24, 26, 37, 39, 42, 45, 49, 62, 63, 74, 83, y 87.
Este código es usado en fragmento 89.

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

4 ⟨*Dependencias de casos de prueba* 4⟩≡
```
(require rackunit)
```
Esta definición se continua en fragmento 15.
Este código es usado en fragmento 89.

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?.

5 ⟨*Definición de abstracción* 5⟩≡
```
(define-struct/contract abstracción
  ([argumento variable?] [cuerpo expresión?])
  #:transparent
  #:mutable)
```
Define:
abstracción, usado en fragmentos 1, 6, 8, 16, 17, 21, 22, 5, 28, 47, 55, 5, 56, 62, 63, 65–67, 77–79, 81, 85, 87, 91, 93, 97–99, 101, 103, y 104.
Hace referencia a expresión? 1 y variable 2.
Este código es usado en fragmento 27.

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

6 ⟨*Casos de prueba* 3⟩+≡
```
(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")))
```
Hace referencia a abstracción 5 y variable 2.

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?`.

7 ⟨*Definición de aplicación* 7⟩≡
```
(define-struct/contract aplicación
  ([operador expresión?] [operando expresión?])
  #:transparent
  #:mutable)
```
Define:
aplicación, usado en fragmentos 1, 8, 12, 16, 17, 23, 24, 7, 28, 47, 7, 48–50, 53, 55, 60, 62, 63, 65–67, 77, 78, 80, 81, 85, 87, 91, 93, 97–99, 101, 103, y 104.
Hace referencia a expresión? 1.
Este código es usado en fragmento 27.

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

8 ⟨*Casos de prueba* 3⟩+≡
```
(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")))
```
Hace referencia a abstracción 5, aplicación 7, y variable 2.

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.

9 ⟨*Definición de hueco* 9⟩≡
```
(define-struct/contract hueco
  ()
  #:transparent)
```
Define:
hueco, usado en fragmentos 1, 10, 17, 26, 9, 52, 9, 53, 62, 63, 65–67, 77, 78, 81, 87, 91, 93, y 101.
Este código es usado en fragmento 27.

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.

10 ⟨*Casos de prueba* 3⟩+≡
```
(test-case "Implementación de huecos"
  (check-true (hueco? (hueco)))
  (check-false (hueco? 42)))
```
Hace referencia a hueco 9.

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?).

11 ⟨*Definición de metainstrucción* 11⟩≡
```
(define-struct/contract metainstrucción
  ([nombre string?] [argumentos (*list/c expresión?)])
  #:transparent
  #:mutable)
```
Define:
metainstrucción, usado en fragmentos 1, 12, 11, 13, 11, 14, 25, 26, 11, 60–63, 65–67, 77, 78, 81, 85, 87, 91, 93, 94, y 97–104.
Hace referencia a expresión? 1.
Esta definición se continua en fragmento 13.
Este código es usado en fragmento 27.

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

12 ⟨*Casos de prueba* 3⟩+≡
```
(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"))))
```
Hace referencia a aplicación 7, metainstrucción 11, metainstrucciones 13, y variable 2.

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 III se muestra la implementación de varias metainstrucciones.

13 ⟨*Definición de metainstrucción* 11⟩+≡
```
(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`, usado en fragmentos 13, 14, 87, 94, y 97–104.
`metainstrucciones`, usado en fragmentos 13, 12–14, 26, 66, 13, 85, 94, y 95.
Hace referencia a `metainstrucción` 11.

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

14 ⟨*Casos de prueba* 3⟩+≡
```
(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))))
```
Hace referencia a `define-metainstrucción` 13, `metainstrucción` 11, `metainstrucciones` 13, y `variable` 2.

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

15 ⟨*Dependencias de casos de prueba* 4⟩+≡
```
(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.

16 ⟨*Definición de término* 16⟩≡
```
(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?`, usado en fragmentos 16–18 y 97.

Hace referencia a abstracción 5, aplicación 7, y variable 2.
Este código es usado en fragmento 27.

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

17 ⟨*Casos de prueba* 3⟩+≡
```
(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))))))
```
Hace referencia a abstracción 5, aplicación 7, hueco 9, término? 16, y variable 2.

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.

18 ⟨*Comparación de expresiones* 18⟩≡
```
(define/contract (expresión=? e1 e2)
  (expresión? expresión? . -> . boolean?)
  (equal? e1 e2))

(define/contract (término=? t1 t2)
  (término? término? . -> . boolean?)
  (equal? t1 t2))
```
Define:
  expresión=?, usado en fragmentos 18, 21, y 23.
  término=?, usado en fragmento 18.
Hace referencia a expresión? 1 y término? 16.
Este código es usado en fragmento 27.

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.

19 ⟨*Comparación de variables* 19⟩≡
```
(define/contract (variable=? v1 v2)
  (variable? variable? . -> . boolean?)
  (eq? v1 v2))

(define/contract (átomo=? v1 v2)
```

```
    (variable? variable? . -> . boolean?)
    (equal? v1 v2))
```

Define:
átomo=?, usado en fragmentos 19–21.
variable=?, usado en fragmentos 19 y 20.
Hace referencia a variable 2.
Este código es usado en fragmento 27.

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

20 ⟨*Casos de prueba* 3⟩+≡

```
(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"))))
```

Hace referencia a átomo=? 19, variable 2, y variable=? 19.

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

21 ⟨*Comparación de abstracciones* 21⟩≡

```
(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=?, usado en fragmento 22.
cuerpo=?, usado en fragmento 22.
Hace referencia a abstracción 5, expresión=? 18, y átomo=? 19.
Este código es usado en fragmento 27.

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

22 ⟨*Casos de prueba* 3⟩+≡

```
(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")))))
```

Hace referencia a abstracción 5, argumento=? 21, cuerpo=? 21, y variable 2.

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

23 ⟨*Comparación de aplicaciones* 23⟩≡
```
(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=?, usado en fragmento 24.
operando=?, usado en fragmento 24.
Hace referencia a aplicación 7 y expresión=? 18.
Este código es usado en fragmento 27.

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

24 ⟨*Casos de prueba* 3⟩+≡
```
(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")))))
```
Hace referencia a aplicación 7, operador=? 23, operando=? 23, y variable 2.

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

25 ⟨*Comparación de metainstrucciones* 25⟩≡
```
(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=?, usado en fragmento 26.

cantidad-argumentos=?, usado en fragmento 26.
nombre=?, usado en fragmento 26.
Hace referencia a metainstrucción 11.
Este código es usado en fragmento 27.

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

26 ⟨*Casos de prueba* 3⟩+≡

```
(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"))))))
```

Hace referencia a argumentos=? 25, cantidad-argumentos=? 25, hueco 9, metainstrucción 11, metainstrucciones 13, nombre=? 25, y variable 2.

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:

27 ⟨*Estructuras de las expresiones* 27⟩≡

⟨*Definición de expresión* 1⟩
⟨*Definición de término* 16⟩
⟨*Comparación de expresiones* 18⟩
⟨*Definición de variable* 2⟩
⟨*Comparación de variables* 19⟩
⟨*Definición de abstracción* 5⟩
⟨*Comparación de abstracciones* 21⟩
⟨*Definición de aplicación* 7⟩
⟨*Comparación de aplicaciones* 23⟩
⟨*Definición de hueco* 9⟩
⟨*Definición de metainstrucción* 11⟩
⟨*Comparación de metainstrucciones* 25⟩

Este código es usado en fragmento 88.

II. 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 λ.

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 $D = \{`(', `)', `[', `]', `.', `,', `\lambda', ` '\}$ 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 Ω 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:

28 ⟨*El combinador Y* 28⟩≡

```
(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"))))))
```

Hace referencia a `abstracción` 5, `aplicación` 7, y `variable` 2.
Fragmento raíz (no usado en este documento).

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

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

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 Ω 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 estructura 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 'λ' 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:

29 ⟨*Sintaxis del lenguaje* 29⟩≡
⟨*Configuración de la sintaxis* 30⟩
⟨*Parseo del lenguaje* 31⟩
⟨*Escritura del lenguaje* 64⟩
Este código es usado en fragmento 88.

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 'λ' con código 955. De igual manera la variable `símbolo-lambda` debe ser el valor del caracter lambda.

30 ⟨*Configuración de la sintaxis* 30⟩≡
```
(define hay-lambda? true)

(define/contract (símbolo-lambda? x)
  (char? . -> . boolean?)
  (and hay-lambda? (char=? (integer->char 955) x)))
```
Define:
`hay-lambda?`, usado en fragmentos 30, 70, y 82.
`símbolo-lambda?`, usado en fragmentos 54 y 57.
Este código es usado en fragmento 29.

II.I. 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:

31 ⟨*Parseo del lenguaje* 31⟩≡
⟨*Procedimientos auxiliares del parser* 38⟩
⟨*Algoritmo del parser* 32⟩
⟨*Punto de entrada del parser* 33⟩
Este código es usado en fragmento 29.

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.

32 ⟨*Algoritmo del parser* 32⟩≡
```
(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 35⟩))
```
Define:
`parsear-expresión`, usado en fragmentos 32, 33, 37, 42, 32, 44, 45, 32, 47, 32, 50, 32, 53, 32, 55, 56, 60, y 61.
Hace referencia a `expresión?` 1 y `parsear` 33.
Este código es usado en fragmento 31.

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.

33 ⟨*Punto de entrada del parser* 33⟩≡
```
(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`, usado en fragmentos 33, 32–34, 37, 42, 33, 44, 45, 33, 47, 33, 50, 33, 52, 33, 53, 33, 55, 33, 56, 33, 59, 60, 33, 61–63, 74, 83, 33, 86, 87, 90, 95, y 102.
Hace referencia a `expresión?` 1 y `parsear-expresión` 32.
Esta definición se continua en fragmento 34.
Este código es usado en fragmento 31.

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.

34 ⟨*Punto de entrada del parser* 33⟩+≡
```
(define/contract (parsear-cadena s)
  (string? . -> . (or/c expresión? eof-object?))
  (parsear (open-input-string s)))
```
Define:
`parsear-cadena`, usado en fragmentos 34, 37, 42, 45, 62, 63, 74, 83, 34, 86, 87, 90, 95, y 102.
Hace referencia a `expresión?` 1 y `parsear` 33.

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`.

35 ⟨*Casos del algoritmo de parseo* 35⟩≡
```
[(eof-object? c)
 ⟨Se leyó un eof-object 36⟩]
```
Esta definición se continua en fragmentos 40, 43, 46, 51, 54, y 58.
Este código es usado en fragmento 32.

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.

36 ⟨*Se leyó un* `eof-object` 36⟩≡
```
(if (null? a) c a)
```
Este código es usado en fragmento 35.

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

37 ⟨*Casos de prueba* 3⟩+≡
```
(test-case "Lectura de fin de archivo"
  (check-equal? (parsear-cadena "") eof)
  (check-equal? (parsear-expresión (open-input-string "") eof (variable "x"))
                (variable "x")))
```
Hace referencia a `parsear` 33, `parsear-cadena` 34, `parsear-expresión` 32, y `variable` 2.

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.

38 ⟨*Procedimientos auxiliares del parser* 38⟩≡
```
(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?`, usado en fragmentos 38–40 y 57.
Esta definición se continua en fragmentos 47, 48, 52, 56, 57, 59, y 61.
Este código es usado en fragmento 31.

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

39 ⟨*Casos de prueba* 3⟩+≡
```
(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))))
```
Hace referencia a delimitador-final? 38.

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

40 ⟨*Casos del algoritmo de parseo* 35⟩+≡
```
[(delimitador-final? c)
 ⟨Se leyó un delimitador final 41⟩]
```
Hace referencia a delimitador-final? 38.

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.

41 ⟨*Se leyó un delimitador final* 41⟩≡
```
(if (null? a) (read-char p) a)
```
Este código es usado en fragmento 40.

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

42 ⟨*Casos de prueba* 3⟩+≡
```
(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) #\)))
```
Hace referencia a parsear 33, parsear-cadena 34, parsear-expresión 32, y variable 2.

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.

43 ⟨*Casos del algoritmo de parseo* 35⟩+≡
```
[(char-whitespace? c)
 ⟨Se leyó un espacio en blanco 44⟩]
```

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

44 ⟨*Se leyó un espacio en blanco* 44⟩≡
```
(read-char p)
(parsear-expresión p (peek-char p) a)
```
Hace referencia a parsear 33 y parsear-expresión 32.
Este código es usado en fragmento 43.

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

45 ⟨*Casos de prueba* 3⟩+≡
```
(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"))))
```
Hace referencia a parsear 33, parsear-cadena 34, parsear-expresión 32, y variable 2.

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.

46 ⟨*Casos del algoritmo de parseo* 35⟩+≡
```
[(char=? #\( c)
 ⟨Se leyó un paréntesis abierto 50⟩]
```

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.

47 ⟨*Procedimientos auxiliares del parser* 38⟩+≡
```
(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, usado en fragmentos 47 y 50.
Hace referencia a abstracción 5, aplicación 7, parsear 33, y parsear-expresión 32.

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.

48 ⟨*Procedimientos auxiliares del parser* 38⟩+≡
```
(define/contract (aplicación/identidad a e)
  ((or/c expresión? null?) expresión? . -> . expresión?)
  (if (null? a) e (aplicación a e)))
```
Define:
aplicación/identidad, usado en fragmentos 48–50, 53, 55, y 60.
Hace referencia a aplicación 7 y expresión? 1.

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

49 ⟨*Casos de prueba* 3⟩+≡
```
(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"))))
```
Hace referencia a aplicación 7, aplicación/identidad 48, y variable 2.

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

50 ⟨*Se leyó un paréntesis abierto* 50⟩≡
```
(read-char p)
(let ([e (parsear-paréntesis p)])
  (parsear-expresión p (peek-char p) (aplicación/identidad a e)))
```
Hace referencia a aplicación 7, aplicación/identidad 48, parsear 33, parsear-expresión 32, y parsear-paréntesis 47.
Este código es usado en fragmento 46.

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 '['.

51 ⟨*Casos del algoritmo de parseo* 35⟩+≡
```
[(char=? #\[ c)
 ⟨Se leyó un corchete abierto 53⟩]
```

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.

52 ⟨*Procedimientos auxiliares del parser* 38⟩+≡
```
(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, usado en fragmentos 52 y 53.
Hace referencia a hueco 9 y parsear 33.

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.

53 ⟨*Se leyó un corchete abierto* 53⟩≡
```
(read-char p)
(let ([e (parsear-hueco p)])
  (parsear-expresión p (peek-char p) (aplicación/identidad a e)))
```
Hace referencia a aplicación 7, aplicación/identidad 48, hueco 9, parsear 33, parsear-expresión 32, y parsear-hueco 52.
Este código es usado en fragmento 51.

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 'λ'.

54 ⟨*Casos del algoritmo de parseo* 35⟩+≡
```
[(or (símbolo-lambda? c) (char=? #\\ c))
 ⟨Se leyó una lambda 55⟩]
```
Hace referencia a símbolo-lambda? 30.

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.

55 ⟨*Se leyó una lambda* 55⟩≡
```
(read-char p)
(let ([e (parsear-abstracción p)])
  (parsear-expresión p (peek-char p) (aplicación/identidad a e)))
```
Hace referencia a abstracción 5, aplicación 7, aplicación/identidad 48, parsear 33, parsear-abstracción 56, y parsear-expresión 32.
Este código es usado en fragmento 54.

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.

56 ⟨*Procedimientos auxiliares del parser* 38⟩+≡
```
(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, usado en fragmentos 55 y 56.
Hace referencia a abstracción 5, delimitador? 57, expresión? 1, parsear 33, parsear-expresión 32, parsear-nombre 59, y variable 2.

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.

57 ⟨*Procedimientos auxiliares del parser* 38⟩+≡

```
(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?, usado en fragmentos 56, 57, y 59.
Hace referencia a delimitador-final? 38 y símbolo-lambda? 30.

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.

58 ⟨*Casos del algoritmo de parseo* 35⟩+≡

```
[else
 ⟨Se leyó un constituyente 60⟩]
```

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.

59 ⟨*Procedimientos auxiliares del parser* 38⟩+≡

```
(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, usado en fragmentos 56, 59, y 60.
Hace referencia a delimitador? 57, longitud 97, y parsear 33.

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.

60 ⟨*Se leyó un constituyente* 60⟩≡
```
((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)])))
```
Hace referencia a aplicación 7, aplicación/identidad 48, metainstrucción 11, parsear 33, parsear-corchetes 61, parsear-expresión 32, parsear-nombre 59, y variable 2.
Este código es usado en fragmento 58.

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

61 ⟨*Procedimientos auxiliares del parser* 38⟩+≡
```
(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, usado en fragmentos 60 y 61.
Hace referencia a expresión? 1, metainstrucción 11, parsear 33, y parsear-expresión 32.

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.

62 ⟨*Casos de prueba* 3⟩+≡
```
(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)))
```

Hace referencia a abstracción 5, aplicación 7, hueco 9, metainstrucción 11, parsear 33, parsear-cadena 34, y variable 2.

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.

63 ⟨*Casos de prueba* 3⟩+≡

```
(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")))))
```

Hace referencia a abstracción 5, aplicación 7, hueco 9, metainstrucción 11, parsear 33, parsear-cadena 34, y variable 2.

II.II. 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:

64 ⟨*Escritura del lenguaje* 64⟩≡

```
⟨Algoritmos principales de escritura 65⟩
⟨Escritores usuales 68⟩
⟨Escritores misceláneos 75⟩
```

Este código es usado en fragmento 29.

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.

65 ⟨*Algoritmos principales de escritura* 65⟩≡

```
(define/contract (escritor-formal e-variable e-abstracción e-aplicación
                                  e-hueco e-metainstrucción)
  ⟨Contrato para escritores formales 66⟩
  ⟨Algoritmo de escritura formal 67⟩)
```

Define:
escritor-formal, usado en fragmentos 65, 68, 75, 77, 65, y 93.
Hace referencia a abstracción 5, aplicación 7, hueco 9, metainstrucción 11, y variable 2.
Esta definición se continua en fragmento 78.
Este código es usado en fragmento 64.

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.

66 ⟨*Contrato para escritores formales* 66⟩≡
```
((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
```
Hace referencia a `abstracción` 5, `aplicación` 7, `expresión?` 1, `hueco` 9, `metainstrucción` 11, `metainstrucciones` 13, y `variable` 2.
Este código es usado en fragmento 65.

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

67 ⟨*Algoritmo de escritura formal* 67⟩≡
```
(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))]))
```
Hace referencia a `abstracción` 5, `aplicación` 7, `hueco` 9, `metainstrucción` 11, y `variable` 2.
Este código es usado en fragmento 65.

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.

68 ⟨*Escritores usuales* 68⟩≡
```
(define escribir-expresión-formal
  (escritor-formal
   ⟨Escribir formalmente variable a texto plano 69⟩
   ⟨Escribir formalmente abstracción a texto plano 70⟩
   ⟨Escribir formalmente aplicación a texto plano 71⟩
   ⟨Escribir formalmente hueco a texto plano 72⟩
   ⟨Escribir formalmente metainstrucción a texto plano 73⟩))
```
Define:
`escribir-expresión-formal`, usado en fragmentos 70, 68, 71, 73, 74, 68, 90, y 92.
Hace referencia a `escritor-formal` 65.
Esta definición se continua en fragmento 82.
Este código es usado en fragmento 64.

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.

69 ⟨*Escribir formalmente variable a texto plano* 69⟩≡
```
(lambda (e nombre) nombre)
```
Este código es usado en fragmento 68.

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 'λ' se debe verificar si se puede usar su código en `Unicode`, de lo contrario usar el caracter '`\`'.

70 ⟨*Escribir formalmente abstracción a texto plano* 70⟩≡
```
(lambda (e argumento cuerpo)
  (format "(~a~a.~a)"
          (if hay-lambda? (integer->char 955) #\\)
          (escribir-expresión-formal argumento)
          (escribir-expresión-formal cuerpo)))
```
Hace referencia a `escribir-expresión-formal` 68 y `hay-lambda?` 30.
Este código es usado en fragmento 68.

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

71 ⟨*Escribir formalmente aplicación a texto plano* 71⟩≡
```
(lambda (e operador operando)
  (format "(~a ~a)"
          (escribir-expresión-formal operador)
          (escribir-expresión-formal operando)))
```
Hace referencia a `escribir-expresión-formal` 68.
Este código es usado en fragmento 68.

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

72 ⟨*Escribir formalmente hueco a texto plano* 72⟩≡
```
(lambda (e) "[ ]")
```
Este código es usado en fragmento 68.

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

73 ⟨*Escribir formalmente metainstrucción a texto plano* 73⟩≡
```
(lambda (e nombre argumentos)
  (format "~a[~a]" nombre
          (apply string-append
                 (add-between (map escribir-expresión-formal argumentos)
                              ", "))))
```
Hace referencia a `escribir-expresión-formal` 68.
Este código es usado en fragmento 68.

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

74 ⟨*Casos de prueba* 3⟩+≡
```
(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))))
```

Hace referencia a escribir-expresión-formal 68, parsear 33, y parsear-cadena 34.

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 'λ' como \lambda, espacios pequeños como \, y espacios grandes como \quad.

75 ⟨*Escritores misceláneos* 75⟩≡

```
(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, usado en fragmento 75.
Hace referencia a escritor-formal 65.
Esta definición se continua en fragmentos 76, 77, y 84.
Este código es usado en fragmento 64.

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.

76 ⟨*Escritores misceláneos* 75⟩+≡

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

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

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

```
\end{document}

DOCUMENTO-LATEX
  )
```

Define:
plantilla-latex, usado en fragmento 76.

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.

77 ⟨*Escritores misceláneos* 75⟩+≡

```
(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, usado en fragmento 77.
Hace referencia a abstracción 5, aplicación 7, escritor-formal 65, hueco 9, metainstrucción 11, y variable 2.

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.

78 ⟨*Algoritmos principales de escritura* 65⟩+≡
```
(define/contract (escritor-breve e-variable e-abstracción e-aplicación
                                 e-hueco e-metainstrucción)
  ((variable? string? . -> . any/c)
   ⟨Contrato para escritor formal de abstracciones 79⟩
   ⟨Contrato para escritor formal de aplicaciones 80⟩
   (hueco? . -> . any/c)
   (metainstrucción? string? (*list/c expresión?) . -> . any/c)
   . -> . (expresión? . -> . any/c))
  ⟨Algoritmo de escritura breve 81⟩)
```
Define:
escritor-breve, usado en fragmentos 78, 82, 84, 91, y 93.
Hace referencia a abstracción 5, aplicación 7, expresión? 1, hueco 9, metainstrucción 11, y variable 2.

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.

79 ⟨*Contrato para escritor formal de abstracciones* 79⟩≡
```
(abstracción? (*list/c variable? variable?) expresión?
              . -> . any/c)
```
Hace referencia a abstracción 5, expresión? 1, y variable 2.
Este código es usado en fragmento 78.

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.

80 ⟨*Contrato para escritor formal de aplicaciones* 80⟩≡

```
(aplicación? (*list/c expresión? expresión? expresión?)
             (*list/c boolean? boolean? boolean?)
             . -> . any/c)
```

Hace referencia a aplicación 7 y expresión? 1.
Este código es usado en fragmento 78.

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.

81 ⟨*Algoritmo de escritura breve* 81⟩≡

```
(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))]))
```

Hace referencia a abstracción 5, aplicación 7, hueco 9, metainstrucción 11, y variable 2.
Este código es usado en fragmento 78.

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.

82 ⟨*Escritores usuales* 68⟩+≡

```
(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, usado en fragmentos 83 y 92.
Hace referencia a escritor-breve 78 y hay-lambda? 30.

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

83 ⟨*Casos de prueba* 3⟩+≡

```
(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))))
```

Hace referencia a escribir-expresión-breve 82, parsear 33, y parsear-cadena 34.

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.

84 ⟨*Escritores misceláneos* 75⟩+≡

```
(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, usado en fragmento 84.
Hace referencia a escritor-breve 78.

III. 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.

85 ⟨*Evaluación de expresiones* 85⟩≡

```
(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, usado en fragmentos 86 y 92.
Hace referencia a abstracción 5, aplicación 7, expresión? 1, metainstrucción 11, y metainstrucciones 13.
Esta definición se continua en fragmento 86.
Este código es usado en fragmento 88.

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

86 ⟨*Evaluación de expresiones* 85⟩+≡

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

Define:
evaluar-cadena, usado en fragmentos 86 y 87.
Hace referencia a evaluar-expresión 85, parsear 33, y parsear-cadena 34.

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

87 ⟨*Casos de prueba* 3⟩+≡

```
(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")))
```

Hace referencia a abstracción 5, aplicación 7, define-metainstrucción 13, evaluar-cadena 86, hueco 9, metainstrucción 11, parsear 33, parsear-cadena 34, y variable 2.

IV. 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.

88 ⟨*lambda-lib.rkt* 88⟩≡

```
#lang racket
(provide (all-defined-out))
⟨Estructuras de las expresiones 27⟩
⟨Sintaxis del lenguaje 29⟩
⟨Evaluación de expresiones 85⟩
```

Fragmento raíz (no usado en este documento).

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.

89 ⟨*pruebas-lambda-lib.rkt* 89⟩≡

```
#lang racket
⟨Dependencias de casos de prueba 4⟩
(require "lambda-lib.rkt")
⟨Casos de prueba 3⟩
```

Fragmento raíz (no usado en este documento).

CAPÍTULO II

ENTORNO INTERACTIVO

El sistema `Lambda` puede ser utilizado como biblioteca para otros programas que requieran trabajar con el cálculo λ, 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 λ 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 λ.

I. Editores

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

I.I. Editor de texto

90 ⟨*Editor de texto* 90⟩≡

```
(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%, usado en fragmento 95.
Hace referencia a escribir-expresión-formal 68, parsear 33, y parsear-cadena 34.
Este código es usado en fragmento 96.

I.II. Editor estructural

91 ⟨*Editor estructural* 91⟩≡

```
(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")
          '(#\u3b1 #\u3b2 #\u3c8 #\u3b4 #\u3b5 #\u3c6 #\u3b3 #\u3b7 #\u3b9 #\u3be
                   #\u3ba  #\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 ("eliminar-texto" editor evento) "backspace"
       (send editor elimina))
(atajo ("enviar-expresión" editor evento) "c:enter"
       (enviar-expresión (send editor obtener-expresión)))))
```

Define:

editor-estructural%, usado en fragmento 95.
escalar-dibujo, usado en fragmento 93.
Hace referencia a abstracción 5, aplicación 7, escritor-breve 78, hueco 9, metainstrucción 11, y variable 2.
Este código es usado en fragmento 96.

II. Historial

92 ⟨*Historial* 92⟩≡

```
(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%, usado en fragmento 95.
expresión-snip%, nunca usado.
historial%, usado en fragmento 95.
Hace referencia a escribir-expresión-breve 82, escribir-expresión-formal 68, y evaluar-expresión 85.
Este código es usado en fragmento 96.

III. Visualizador

93 ⟨*Visualizador* 93⟩≡

```
(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%, usado en fragmento 95.
figura-escrita-breve-lienzo%, usado en fragmento 95.
figura-escrita-formal-lienzo%, usado en fragmento 95.
Hace referencia a abstracción 5, aplicación 7, escalar-dibujo 91, escritor-breve 78, escritor-formal 65, hueco 9, metainstrucción 11, y variable 2.
Este código es usado en fragmento 96.

IV. Registro de metainstrucciones

94 ⟨*Registro de metainstrucciones* 94⟩≡

```
(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)))))
```

Hace referencia a `define-metainstrucción` 13, `metainstrucción` 11, y `metainstrucciones` 13.
Este código es usado en fragmento 96.

V. Interfaz gráfica

95 ⟨*Uniendo las componentes de la interfaz gráfica* 95⟩≡

```
;; 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)
```

Hace referencia a `código-lienzo%` 93, `editor-estructural%` 91, `editor-texto%` 90, `evaluación%` 92, `figura-escrita-breve-lienzo%` 93, `figura-escrita-formal-lienzo%` 93, `historial%` 92, `metainstrucciones` 13, `parsear` 33, y `parsear-cadena` 34.
Este código es usado en fragmento 96.

VI. Código fuente

96 ⟨*lambda-gui.rkt* 96⟩≡

```
#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 90⟩
⟨Editor estructural 91⟩
⟨Historial 92⟩
⟨Visualizador 93⟩
⟨Registro de metainstrucciones 94⟩
⟨Uniendo las componentes de la interfaz gráfica 95⟩
```

Fragmento raíz (no usado en este documento).

CAPÍTULO III

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.

I. Análisis de términos

97 ⟨*Metainstrucciones* 97⟩≡

```
(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`, usado en fragmento 59.
Hace referencia a `abstracción` 5, `aplicación` 7, `define-metainstrucción` 13, `metainstrucción` 11, `término?` 16, y `variable` 2.
Esta definición se continua en fragmentos 98–104.
Este código es usado en fragmento 105.

98 ⟨*Metainstrucciones* 97⟩+≡

```
(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`, usado en fragmentos 99 y 100.
Hace referencia a `abstracción` 5, `aplicación` 7, `define-metainstrucción` 13, `metainstrucción` 11, y `variable` 2.

II. Transformación de términos

99 ⟨*Metainstrucciones* 97⟩+≡
```
(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 usado.
Hace referencia a abstracción 5, aplicación 7, define-metainstrucción 13, metainstrucción 11, variable 2, variable-ajena 100, y variables-libres 98.

100 ⟨*Metainstrucciones* 97⟩+≡
```
(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, usado en fragmento 99.
Hace referencia a define-metainstrucción 13, metainstrucción 11, variable 2, y variables-libres 98.

101 ⟨*Metainstrucciones* 97⟩+≡
```
(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 usado.
Hace referencia a abstracción 5, aplicación 7, define-metainstrucción 13, hueco 9, metainstrucción 11, y variable 2.

III. Codificaciones

102 ⟨*Metainstrucciones* 97⟩+≡
```
(define-metainstrucción (T)
  (parsear-cadena "\\x y.x"))

(define-metainstrucción (F)
  (parsear-cadena "\\x y.y"))
```
Define:
F, nunca usado.
T, nunca usado.
Hace referencia a define-metainstrucción 13, metainstrucción 11, parsear 33, y parsear-cadena 34.

103 ⟨*Metainstrucciones* 97⟩+≡
```
(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 usado.
Hace referencia a abstracción 5, aplicación 7, define-metainstrucción 13, metainstrucción 11, y variable 2.

104 ⟨*Metainstrucciones* 97⟩+≡
```
(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 usado.
Hace referencia a abstracción 5, aplicación 7, define-metainstrucción 13, metainstrucción 11, y variable 2.

105 ⟨*lambda-init.rkt* 105⟩≡
⟨*Metainstrucciones* 97⟩
Fragmento raíz (no usado en este documento).