Lambda

Un programa para la exploración del cálculo λ puro

Introducción El programa `lambda`[1] tiene la finalidad de proveer una entorno computacional ameno para estudiar el contenido de este trabajo de manera interactiva. La presente implementación está escrita en el lenguaje de programación `Racket`[2] y consta de un intérprete de expresiones del cálculo λ el cuál puede ser utilizado de tres maneras diferentes:

1. A través de un ciclo de lectura, evaluación y escritura (conocido como *REPL*) por consola;
2. Por medio de un editor estructural gráfico;
3. Como una biblioteca para facilitar la programación de algoritmos que involucren términos del cálculo λ.

El corazón del intérprete es un *evaluador*, una función que transforma expresiones de un lenguaje a valores, los cuales a su vez pueden ser expresiones de este lenguaje. El lenguaje con el que trabaja este procedimiento es una extensión a Λ que incluye "huecos" y "metainstrucciones", estas últimas permiten manipular términos λ de manera arbitraria, por ejemplo calcular la longitud de un término M involucraría invocar una metainstrucción similar a la definición de $\|M\|$. Las expresiones de este lenguaje son almacenadas en memoria como un árbol de sintaxis. Para construír estos árboles se utilizan dos métodos: usando un *lector* (o parser) que transforma cadenas de caracteres a árboles o utilizando el *editor estructural* que permite únicamente editar expresiones válidas del lenguaje.

Estructuras del lenguaje

Árbol de sintaxis Un árbol de sintaxis para el lenguaje de `lambda` consiste de vértices que pueden ser *variables*, *aplicaciones*, *abstracciones*, *huecos* o *metainstrucciones*. Estos tipos de vértices son implementados haciendo uso del macro `struct` el cuál a partir de un identificador de la estructura y los identificadores de sus componentes produce un constructor, un predicado y selectores para cada componente.

La estructura que representa a las variables de un árbol es llamada `variable` y se compone de una cadena de caracteres llamada `nombre`.

[1] Nombrado así en ausencia de originalidad.
[2] Para más información, visitar el sitio oficial http://racket-lang.org/.

1 ⟨*estructuras del lenguaje* 1⟩≡
```
(struct variable (nombre) #:transparent)
```

Define:
variable, usado en fragmentos 1, 4, 7, 1, 8, 9, 26, 29, 33, 1, 34–36, 40, 1, 41, y 42.
Esta definición se continua en fragmentos 2, 3, 5, y 6.
Este código es usado en fragmento 10.

La estructura que representa a las abstracciones de un árbol es llamada abstracción, se compone de una variable llamada argumento y de otro vértice del árbol llamado cuerpo, este último puede ser de cualquiera de los cuatro tipos de vértices mencionados.

2 ⟨*estructuras del lenguaje* 1⟩+≡
```
(struct abstracción (argumento cuerpo) #:transparent)
```

Define:
abstracción, usado en fragmentos 2, 4, 7, 2, 8, 9, 22, 2, 25, 2, 26, 33, 36, 40, 42, y 43.

La estructura que representa a las aplicaciones de un árbol es llamada aplicación, se compone de un vértice llamado operador y de otro vértice llamado operando.

3 ⟨*estructuras del lenguaje* 1⟩+≡
```
(struct aplicación (operador operando) #:transparent)
```

Define:
aplicación, usado en fragmentos 3, 4, 7, 3, 8, 9, 20, 3, 21–23, 25, 3, 30, 33, 37, 40, 3, y 43.

Se define el predicado término? para determinar si un objeto dado es un término λ, es decir, si es una estructura variable, abstracción o aplicación. Se debe verificar que los campos de cada subtérmino corresponden a un término λ adecuado, por ejemplo, las abstracciones deben tener como argumento una variable y como cuerpo un término.

4 ⟨*predicados del lenguaje* 4⟩≡
```
(define (término? x)
  (cond
   [(variable? x)    (string? (variable-nombre x))]
   [(abstracción? x) (and (variable? (abstracción-argumento x))
                          (término? (abstracción-cuerpo x)))]
   [(aplicación? x)  (and (término? (aplicación-operador x))
                          (término? (aplicación-operando x)))]
   [else             #f]))
```

Define:
término?, usado en fragmentos 4 y 9.
Hace referencia a abstracción 2, aplicación 3, y variable 1.
Esta definición se continua en fragmento 7.
Este código es usado en fragmento 10.

La estructura que representa a las metainstrucciones de un árbol es llamada `metainstrucción`, se compone de una cadena de caracteres llamada `nombre` y de una lista de vértices llamada `argumentos`.

5 ⟨*estructuras del lenguaje* 1⟩+≡
```
(struct metainstrucción (nombre argumentos) #:transparent)
```

Define:
`metainstrucción`, usado en fragmentos 5, 7, 9, 11, 29, 33, 39, 40, y 45.

Para representar un vértice sin un término asignado se utiliza la estructura `hueco`, esta funciona como una especie de comodín o valor nulo.

6 ⟨*estructuras del lenguaje* 1⟩+≡
```
(struct hueco () #:transparent)
```

Define:
`hueco`, usado en fragmentos 6, 7, 9, 23, 24, 33, 40, y 44.

De manera análoga a la identificación de términos λ mediante el procedimiento `término?` se define el predicado `expresión?` el cuál permite distinguir estructuras que sean expresiones de este lenguaje, incluyendo a los términos λ.

7 ⟨*predicados del lenguaje* 4⟩+≡
```
(define (expresión? x)
  (cond
   [(variable? x)        (string? (variable-nombre x))]
   [(abstracción? x)     (and (variable? (abstracción-argumento x))
                              (expresión? (abstracción-cuerpo x)))]
   [(aplicación? x)      (and (expresión? (aplicación-operador x))
                              (expresión? (aplicación-operando x)))]
   [(metainstrucción? x) (and (string? (metainstrucción-nombre x))
                              (every expresión? (metainstrucción-argumentos x)))]
   [(hueco? x)           #t]
   [else                 #f]))
```

Define:
`expresión?`, usado en fragmentos 7, 9, y 26.
Hace referencia a `abstracción` 2, `aplicación` 3, `hueco` 6, `metainstrucción` 5, y `variable` 1.

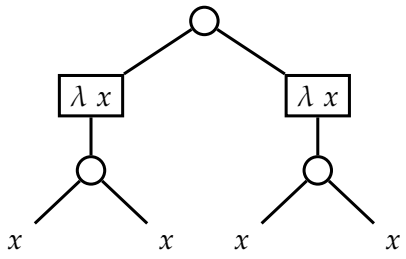
Utilizando los constructores `variable`, `abstracción` y `aplicación`, un término λ como

$$(\lambda x.(x\ x))(\lambda x.(x\ x))$$

puede ser representado en memoria como

8 ⟨*construcción de un término* 8⟩≡
```
(aplicación (abstracción (variable "x") (aplicación (variable "x") (variable "x")))
            (abstracción (variable "x") (aplicación (variable "x") (variable "x"))))
```
Hace referencia a `abstracción` 2, `aplicación` 3, y `variable` 1.
Fragmento raíz (no usado en este documento).

el cuál corresponde a un árbol de sintaxis visualizado graficamente como



Módulo de estructuras del lenguaje Las estructuras y procedimientos del árbol de sintaxis son agrupados en un módulo de `Racket` el cuál podrá ser accesado desde otros módulos. Para "exportar" los identificadores definidos se utiliza el macro `provide`.

9 ⟨*provee las definiciones del árbol* 9⟩≡
```
(provide variable variable? variable-nombre
         abstracción abstracción? abstracción-argumento abstracción-cuerpo
         aplicación aplicación? aplicación-operador aplicación-operando
         metainstrucción metainstrucción? metainstrucción-nombre metainstrucción-argumentos
         hueco hueco?
         término? expresión?)
```
Hace referencia a `abstracción` 2, `aplicación` 3, `expresión?` 7, `hueco` 6, `metainstrucción` 5, `término?` 4, y `variable` 1.
Este código es usado en fragmento 10.

El archivo `estructuras.rkt` contiene este módulo y tiene la siguiente forma:

10 ⟨*estructuras.rkt* 10⟩≡
```
;;; -*- mode: racket; coding: utf-8 -*-
;;; Copyright (C) 2016 Eduardo Acuña Yeomans <eduardo.acye@gmail.com>
;;; Procura no editar este archivo, mejor modifica lambda.nw

#lang racket/base

(require srfi/1)

⟨provee las definiciones del árbol 9⟩

;;; Estructuras del árbol de sintaxis
⟨estructuras del lenguaje 1⟩

;;; Predicados especiales
```

⟨*predicados del lenguaje* 4⟩
Fragmento raíz (no usado en este documento).

Lectura de expresiones

Representación textual El lenguaje de `lambda` es descrito conceptualmente por las estructuras de los árboles de sintaxis. Sin embargo es tedioso definir expresiones del lenguaje escribiendo manualmente los constructores de las expresiones. Trabajando en un contexto meramente textual como en el código de un lenguaje de programación es conveniente tener una manera de transformar una cadena de caracteres como "$(\lambda x.(xx))(\lambda x.(xx))$" (o mejor aún, con abuso de notación "$(\lambda x.xx)\lambda x.xx$") a un árbol de sintaxis de este término. Este es el trabajo del lector.

La representación textual del lenguaje contempla la escritura de términos descrita en la definición formal de Λ y los abusos de notación contemplados en el trabajo. Adicionalmente se reprsentan las metainstrucciones de tal manera que

11 ⟨*pseudocódigo de una metainstrucción* 11⟩≡
```
(metainstrucción nombre (list argumento-0 argumento-1 ... argumento-n))
```
Hace referencia a `metainstrucción` 5.
Fragmento raíz (no usado en este documento).

tiene como representación textual
`"nombre[argumento-0, argumento-1, ... , argumento-n]"`

Parser El lector de expresiones de este lenguaje es un *parser* descendente recursivo que a partir de un puerto de entrada de `Racket` construye un árbol de sintaxis.

El procedimiento `leer` es el punto de entrada del parser, recibe opcionalmente un puerto llamado `p` que por defecto es `(current-input-port)`.

12 ⟨*lector de expresiones* 12⟩≡
```
(define (leer [p (current-input-port)])
  (let ([e (parsear-expresión p)])
    (cond
     [(and (char? e) (char=? #\; e))
      (leer p)]
     [(not (char? e))
      e]
     [else
      (error 'leer "Se esperaba una expresión completa pero se leyó ‘~a’" e)])))
```

Define:
`leer`, usado en fragmentos 12, 31, 12, y 32.
Hace referencia a `parsear-expresión` 13.
Esta definición se continua en fragmento 13.
Este código es usado en fragmento 32.

El procedimiento `parsear-expresión` se encarga de regresar una expresión `e`, en caso que la estructura sintáctica analizada no haya sido una expresión válida se regresa un caracter que indica el inicio de lo que no se pudo leer. En esta implementación del lector el caracter punto y coma delimita expresiones, por lo tanto, obtener como resultado `#\;` provoca que se analice la siguiente expresión del puerto.

La implementación de `parsear-expresión` considera un puerto de entrada `p` de donde se consumen caracteres, un caracter `c` que es el primer caracter del puerto y un acumulador de expresiones `a` que representa el árbol leído. A partir de `c` se puede determinar el tipo de expresión que se está leyendo.

13 ⟨*lector de expresiones* 12⟩+≡

```
(define (parsear-expresión [p (current-input-port)]
                           [c (peek-char p)]
                           [a null])
  (cond
   [(eof-object? c)
    ⟨lectura de fin de archivo 14⟩]
   [(delimitador-final? c)
    ⟨lectura de delimitador final 16⟩]
   [(char-whitespace? c)
    ⟨lectura de espacio en blanco 19⟩]
   [(char=? #\( c)
    ⟨lectura de paréntesis abierto 20⟩]
   [(char=? #\[ c)
    ⟨lectura de hueco 23⟩]
   [(or (char=? (integer->char 955) c)
        (char=? #\\ c))
    ⟨lectura de lambda 25⟩]
   [else
    ⟨lectura de variable o metainstrucción 30⟩]))
```

Define:
`parsear-expresión`, usado en fragmentos 12, 13, 19, 20, 13, 22, 23, 25, 26, 30, y 31.
Hace referencia a `delimitador-final?` 15.

El análisis de casos tiene tres cláusulas "triviales": cuando no hay más que leer del puerto, cuando se lee un delimitador final y cuando se lee un espacio en blanco. Cuando el siguiente objeto del puerto de entrada es el "fin de archivo", el lector debe regresar este objeto si el acumulador a es núlo, de lo contrario se regresa la expresión acumulada.

14 ⟨*lectura de fin de archivo* 14⟩≡

```
(if (null? a) c a)
```

Este código es usado en fragmento 13.

El siguiente caso es la lectura de un delimitador final, el procedimiento `delimitador-final?` se encarga de determinar si un caracter termina una expresión.

15 ⟨*procedimientos auxiliares de lectura* 15⟩≡
```
(define (delimitador-final? c)
  (or (eof-object? c)
      (char=? #\] c)
      (char=? #\) c)
      (char=? #\. c)
      (char=? #\, c)
      (char=? #\; c)))
```

Define:
`delimitador-final?`, usado en fragmentos 13, 15, y 18.
Esta definición se continua en fragmentos 17, 18, 21, 22, 24, 26–28, y 31.
Este código es usado en fragmento 32.

Cuando un objeto leído de un puerto de entrada satisface este predicado, el lector consume y regresa el delimitador si el acumulador a es núlo, de lo contrario regresa la expresión acumulada.

16 ⟨*lectura de delimitador final* 16⟩≡
```
(if (null? a) (read-char p) a)
```
Este código es usado en fragmento 13.

El caracter delimitador no es consumido en caso de que a no sea núlo ya que usualmente los delimitadores finales tienen asociado un delimitador inicial con el que deben concordar, por ejemplo, el predicado `delimitador-inicial?` se define[3] como

17 ⟨*procedimientos auxiliares de lectura* 15⟩+≡
```
(define (delimitador-inicial? c)
  (or (char=? #\( c)
      (char=? #\[ c)
      (char=? (integer->char 955) c)
      (char=? #\\ c)))
```

Define:
`delimitador-inicial?`, usado en fragmentos 17 y 18.

En el lenguaje, el delimitador `#\(` debe estar asociado al delimitador `#\)` y de igual manera con `#\[` y `#\]`. Esta verificación se realiza en otra parte del código, pero se debe preservar la información del delimitador leído.

Técnicamente, los caracteres delimitadores del lenguaje son los que satisfacen los predicados `delimitador-inicial?`, `delimitador-final?` y `char-whitespace?`, sin embargo es conveniente definirlos por separado ya que son tratados de manera diferente en la lectura de una expresión.

[3] Se realiza una comparación con `(integer->char 955)` ya que 955 es el valor numérico del caracter λ en Unicode.

Para determinar si algún caracter es delimitador (sin importar el tipo de delimitador que sea) se utiliza el predicado `delimitador?`.

18 ⟨*procedimientos auxiliares de lectura* 15⟩+≡
```
(define (delimitador? c)
  (or (delimitador-final? c)
      (delimitador-inicial? c)
      (char-whitespace? c)))
```

Define:
`delimitador?`, usado en fragmentos 18, 26–28, 18, y 32.
Hace referencia a `delimitador-final?` 15 y `delimitador-inicial?` 17.

El tercer y último caso "trivial" en la lectura es el de espacios en blanco. Al encontrarse un caracter de este tipo se consume del puerto y se llama de manera recursiva `parsear-expresión` de tal manera que el caracter es ignorado y se preserva el valor de la expresión acumulada `a`.

19 ⟨*lectura de espacio en blanco* 19⟩≡
```
(read-char p)
(parsear-expresión p (peek-char p) a)
```

Hace referencia a `parsear-expresión` 13.
Este código es usado en fragmento 13.

El resto de los casos son significativamente más complejos que los primeros tres. Primero se aborda el leer un paréntesis abierto, esto puede ocurrir cuando se desea agrupar una aplicación o una abstracción. El procedimiento `parsear-paréntesis` se encarga de leer una expresión después de un paréntesis abierto y verificar que dicha expresión termina con un paréntesis cerrado.

20 ⟨*lectura de paréntesis abierto* 20⟩≡
```
(read-char p)
(let ([e (parsear-paréntesis p)])
  (parsear-expresión p (peek-char p) (aplicación/identidad a e)))
```

Hace referencia a `aplicación` 3, `aplicación/identidad` 21, `parsear-expresión` 13, y `parsear-paréntesis` 22.
Este código es usado en fragmento 13.

Primero se consume el caracter `#\(` del puerto de entrada, después se llama el procedimiento `parsear-paréntesis` el cuál regresará la expresión `e` delimitada por `#\(` y `#\)`. El procedimiento `parsear-expresión` es llamado de manera recursiva ya que es posible que esta expresión entre paréntesis sea parte de una aplicación escrita con abuso de notación, por ejemplo `(M N)P`. El resultado es acumulado con el procedimiento `aplicación/identidad`, este contempla la posibilidad que la expresión leída sea parte de una expresión de la forma `M(N P)` y se define como

21 ⟨*procedimientos auxiliares de lectura* 15⟩+≡
```
(define (aplicación/identidad a e)
  (if (null? a) e (aplicación a e)))
```

Define:
aplicación/identidad, usado en fragmentos 20, 21, 23, 25, 21, y 30.
Hace referencia a aplicación 3.

Cuando el acumulador a es núlo, el nuevo acumulador es la expresión e, de lo contrario, el nuevo acumulador es la aplicación de a con e.

El procedimiento parsear-paréntesis lee una expresión e, verifica que el resultado es una aplicación o una abstracción y verifica que el caracter después la representación textual de e es un paréntesis cerrado.

22 ⟨*procedimientos auxiliares de lectura* 15⟩+≡
```
(define (parsear-paréntesis [p (current-input-port)]
                            [c (peek-char p)])
  (let ([e (parsear-expresión p c)])
    (unless (or (abstracción? e) (aplicación? e))
      (error 'parsear-paréntesis
             "Se esperaba una abstracción o una aplicación pero se leyó ~a" e))
    (let ([c (read-char p)])
      (unless (char=? #\) c)
        (error 'parsear-paréntesis
               "Se esperaba ‘)’ pero se leyó ‘~a’" c))
      e)))
```

Define:
parsear-paréntesis, usado en fragmentos 22, 20, y 22.
Hace referencia a abstracción 2, aplicación 3, y parsear-expresión 13.

Ahora se aborda el caso de leer el caracter #\[este caracter es utilizado después de leer el nombre de una metainstrucción, sin embargo, cuando se encuentra este caracter por sí solo denota el inicio de un hueco, los cuales se conforman de un corchete abierto y uno cerrado con espacios en blanco opcionales entre los corchetes.

23 ⟨*lectura de hueco* 23⟩≡
```
(read-char p)
(let ([e (parsear-hueco p)])
  (parsear-expresión p (peek-char p) (aplicación/identidad a e)))
```
Hace referencia a aplicación 3, aplicación/identidad 21, hueco 6, parsear-expresión 13, y parsear-hueco 24.
Este código es usado en fragmento 13.

24 ⟨*procedimientos auxiliares de lectura* 15⟩+≡
```
(define (parsear-hueco [p (current-input-port)]
                       [c (peek-char p)])
  (cond [(char-whitespace? c)
         (read-char p)
         (parsear-hueco p (peek-char p))]
```

```
        [(char=? #\] c)
         (read-char p)
         (hueco)]
        [else
         (error 'parsear-hueco
                "Se esperaba `]' pero se leyó `~a'" c)]))
```

Define:
parsear-hueco, usado en fragmento 23.
Hace referencia a hueco 6.

El siguiente caso es leer un caracter que representa a λ, en esta implementación estos caracteres son #\\ y el resultado de `(integer->char 955)` que es el caracter Unicode de la letra lambda minúscula. De manera similar a la lectura de paréntesis abierto, se consume del puerto el caracter leído, se obtiene la abstracción con el procedimiento `parsear-abstracción` y se manejan los abusos de notación de la aplicación.

25 ⟨*lectura de lambda* 25⟩≡

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

Hace referencia a abstracción 2, aplicación 3, aplicación/identidad 21, parsear-abstracción 26, y parsear-expresión 13.
Este código es usado en fragmento 13.

La implementación del procedimiento `parsear-abstracción` es un poco más compleja a la de `parsear-paréntesis`, la representación textual de las abstracciones con el abuso de notación es de la forma `"\ x-1 x-2 ... x-n . M"`, ya que el procedimiento inicia después de consumir el caracter de λ, primero se deben identificar todas las variables antes del punto y regresar una abstracción "descurrificada".

26 ⟨*procedimientos auxiliares de lectura* 15⟩+≡

```
(define (parsear-abstracción [p (current-input-port)]
                             [c (peek-char p)])
  (let itera ([c c]
              [a null])
    (cond [(or (eof-object? c)
               (and (delimitador? c) (not (char-whitespace? c)) (not (char=? #\. c))))
           (error 'parsear-abstracción
                  "Se esperaba una variable pero se leyó `~a'" c)]
          [(char-whitespace? c)
           (read-char p)
           (itera (peek-char p) a)]
          [(char=? #\. c)
           (read-char p)
           (if (null? a)
```

```
                (error 'parsear-abstracción
                       "Se esperaba una variable pero se leyó `~a'" c)
                (let ([e (parsear-expresión p)])
                  (if (expresión? e)
                      (foldr abstracción e (reverse a))
                      (error 'parsear-abstracción
                             "Se esperaba una expresión pero se leyó ~a" e))))]
          [else
           (let ([v (variable (parsear-constituyente p))])
             (itera (peek-char p) (cons v a)))])))
```

Define:
parsear-abstracción, usado en fragmentos 26, 25, y 26.
Hace referencia a abstracción 2, delimitador? 18, expresión? 7, parsear-constituyente 27, parsear-expresión 13, y variable 1.

El procedimiento parsear-constituyente se utiliza para consumir del puerto de entrada una cadena de caracteres constituida por no-delimitadores.

27 ⟨*procedimientos auxiliares de lectura* 15⟩+≡

```
(define (parsear-constituyente [p (current-input-port)]
                               [c (peek-char p)])
  (let itera ([a null]
              [c c])
    (if (delimitador? c)
        (if (null? a)
            (error 'parsear-constituyente
                   "Se esperaba un identificador pero se leyó `~a'" c)
            (list->string (reverse a)))
        (itera (cons (read-char p) a)
               (peek-char p)))))
```

Define:
parsear-constituyente, usado en fragmentos 26, 27, y 29.
Hace referencia a delimitador? 18.

Las cadenas conformadas por caracteres constituyentes son usadas en los nombres de las variables y metainstrucciones, para determinar si una cadena es un "nombre" se puede utilizar el predicado nombre?.

28 ⟨*procedimientos auxiliares de lectura* 15⟩+≡

```
(define (nombre? x)
  (and (string? x)
       (not (string=? x ""))
       (not (any delimitador? (string->list x)))))
```

Define:

nombre?, usado en fragmentos 28 y 32.
Hace referencia a delimitador? 18.

La última cláusula del análisis de casos se encarga de leer variables o metainstrucciones, esto es debido a que ambas inician con la lectura de caracteres constituyentes. Para obtener la variable o la expresión se lee una cadena de caracteres constituyentes del puerto de entrada, si el delimitador de esta cadena es #\[entonces la expresión debe ser una metainstrucción, de lo contrario debe ser una variable.

29 ⟨*obtener variable o metainstrucción 29*⟩≡
```
(let* ([x (parsear-constituyente p c)]
       [c (peek-char p)])
  (cond [(and (char? c) (char=? #\[ c))
         (read-char p)
         (metainstrucción x (parsear-corchetes p))]
        [else
         (variable x)]))
```
Hace referencia a metainstrucción 5, parsear-constituyente 27, parsear-corchetes 31, y variable 1.
Este código es usado en fragmento 30.

Al igual que en el caso del paréntesis abierto y que la λ, se debe acumular este valor utilizando aplicación/identidad.

30 ⟨*lectura de variable o metainstrucción 30*⟩≡
```
((lambda (e)
   (parsear-expresión p (peek-char p) (aplicación/identidad a e)))
 ⟨obtener variable o metainstrucción 29⟩)
```
Hace referencia a aplicación 3, aplicación/identidad 21, y parsear-expresión 13.
Este código es usado en fragmento 13.

En el caso que la expresión termine con el corchete abierto, se utiliza el procedimiento parsear-corchetes para obtener una lista de expresiones separadas por coma y delimitadas por #\]. Es importante tener en cuenta que no se admiten espacios entre el nombre de una metainstrucción y el corchete que denota el inicio de su lista de argumentos.

31 ⟨*procedimientos auxiliares de lectura 15*⟩+≡
```
(define (parsear-corchetes [p (current-input-port)]
                           [c (peek-char p)])
  (let itera ([a null]
              [e (parsear-expresión p c)])
       (cond [(eof-object? e)
              (error 'parsear-corchetes
                     "Se esperaba leer ‘]’ o ‘,’ pero se leyó ~a" e)]
             [(char? e)
              (case e
                [(#\]) (reverse a)]
                [(#\,) (itera a (parsear-expresión p))]
```

```
          [else
           (error 'parsear-corchetes
                  "Se esperaba leer `]' o `,' pero se leyó ~a" e)])]
        [else
         (itera (cons e a) (parsear-expresión p))])))
```

Define:
parsear-corchetes, usado en fragmentos 29 y 31.
Hace referencia a leer 12 y parsear-expresión 13.

Módulo de lector El módulo del lector utiliza el módulo de `estructuras.rkt` y provee el procedimiento `leer`, evitando que los usuarios del parser malutilicen los procedimientos auxiliares de lectura. Adicionalmente se proveen los predicados `delimitador?` y `nombre?` que pueden ser reutilizados en otras partes del programa. El archivo `lector.rkt` contiene este módulo y tiene la siguiente forma:

32 ⟨*lector.rkt* 32⟩≡

```
;;; -*- mode: racket; coding: utf-8 -*-
;;; Copyright (C) 2016 Eduardo Acuña Yeomans <eduardo.acye@gmail.com>
;;; Procura no editar este archivo, mejor modifica lambda.nw

#lang racket/base

(require srfi/1
         "estructuras.rkt")

;;; Exportar punto de entrada del parser
(provide leer
         delimitador? nombre?)

;;; Procedimientos principales
⟨lector de expresiones 12⟩

;;; Procedimientos auxiliares
⟨procedimientos auxiliares de lectura 15⟩
```

Hace referencia a delimitador? 18, leer 12, y nombre? 28.
Fragmento raíz (no usado en este documento).

Pruebas del lector El archivo de pruebas `lector-pruebas.rkt` es utilizado para verificar si la funcionalidad del lector es la esperada. Se omite su definición en este documento pero es distribuído con el código del trabajo.

Escritura de expresiones

Así como se tiene un mecanismo para pasar de la representación externa textual de una expresión a una estructura en memoria, también se tiene un mecanismo para realizar el procedimiento inverso, es decir, escribir la representación externa textual de una estructura en memoria.

Diversidad en la escritura En el caso del lenguaje de expresiones esto es algo complicado ya que hay varias representaciones textuales para una sola expresión, por ejemplo, formalmente correcta o con algún abuso de notación. Por lo tanto, la escritura de expresiones se maneja utilizando diferentes procedimientos que resultan en estilos diversos de escritura.

Una cuestión importante que se debe tener en cuenta es que una misma representación externa textual puede escribirse para ser interpretada de manera diferente, por ejemplo, se pudieran escribir diferentes "backends" de escritura: texto plano, LaTeX, `postscript`, o incluso escribir la estructura en memoria de Racket a otro lenguaje de programación en la que pueda ser representada.

Los escritores que son implementados en el programa son:

- A texto plano sin abuso de notación;
- A texto plano con abuso de notación;
- A LaTeXsin abuso de notación;
- A LaTeXcon abuso de notación;
- A una figura funcional de `Racket` sin abuso de notación;
- A una figura funcional de `Racket` con abuso de notación.

Se abordan estos estilos en orden

De expresiones a texto plano Para traducir una expresión a texto plano se procesa el árbol de manera recursiva, concatenando cadenas de caracteres con la representación externa de las subexpresiones.

Primero define el procedimiento `expresión->texto-plano` el cuál se encarga de identificar la clase de expresión que se desea traducir.

33 ⟨*A texto plano sin abuso* 33⟩≡

```
(define (expresión->texto-plano e)
  (cond
   [(variable? e)
    ⟨Traducir variable a texto plano sin abuso 35⟩]
   [(abstracción? e)
    ⟨Traducir abstracción a texto plano sin abuso 36⟩]
```

```
[(aplicación? e)
 ⟨Traducir aplicación a texto plano sin abuso 37⟩]
[(hueco? e)
 ⟨Traducir hueco a texto plano sin abuso 38⟩]
[(metainstrucción? e)
 ⟨Traducir metainstrucción a texto plano sin abuso 39⟩]
[else
 (error 'expresión->texto-plano
        "La expresión ~a no es válida" e)]))
```

Define:
  expresión->texto-plano, usado en fragmentos 33, 36, 37, 39, 33, y 47.
Hace referencia a abstracción 2, aplicación 3, hueco 6, metainstrucción 5, y variable 1.
Esta definición se continua en fragmento 34.
Este código es usado en fragmento 47.

Para traducir una variable a texto plano sin abuso de notación, basta con obtener el nombre de la variable ya que los nombres de las variables ya son cadenas de caracteres. Sin embargo, la traducción de variables es utilizada en la traducción de abstracciones, por lo tanto es conveniente definir el procedimiento `variable->texto-plano` el cuál corrobora que la expresión es una variable y regresa su nombre.

34 ⟨*A texto plano sin abuso* 33⟩+≡

```
(define (variable->texto-plano e)
  (if (variable? e)
      (variable-nombre e)
      (error 'variable->texto-plano
             "La expresión ~a no es una variable" e)))
```

Define:
  variable->texto-plano, usado en fragmentos 34–36 y 41.
Hace referencia a variable 1.

En `expresión->texto-plano` solo se invoca este procedimiento

35 ⟨*Traducir variable a texto plano sin abuso* 35⟩≡

```
(variable->texto-plano e)
```

Hace referencia a variable 1 y variable->texto-plano 34.
Este código es usado en fragmento 33.

Para traducir una abstracción a texto plano sin abuso de notación, primero se debe traducir su argumento y su cuerpo y después concatenarlos de la forma (λ`argumento.cuerpo`).

36 ⟨*Traducir abstracción a texto plano sin abuso* 36⟩≡

```
(format "(~a~a.~a)"
        (integer->char 955)
        (variable->texto-plano (abstracción-argumento e))
```

```
        (expresión->texto-plano (abstracción-cuerpo e)))
```

Hace referencia a `abstracción` 2, `expresión->texto-plano` 33, `variable` 1, y `variable->texto-plano` 34.
Este código es usado en fragmento 33.

Para traducir una aplicación a texto plano sin abuso de notación, se traduce su operador y su operando y después se concatenan de la forma (`operador operando`).

37 ⟨*Traducir aplicación a texto plano sin abuso* 37⟩≡

```
(format "(~a ~a)"
        (expresión->texto-plano (aplicación-operador e))
        (expresión->texto-plano (aplicación-operando e)))
```

Hace referencia a `aplicación` 3 y `expresión->texto-plano` 33.
Este código es usado en fragmento 33.

La traducción de un hueco a texto plano sin abuso de notación es simplemente la cadena compuesta por el corchete abierto y el corchete cerrado.

38 ⟨*Traducir hueco a texto plano sin abuso* 38⟩≡

```
"[ ]"
```

Este código es usado en fragmento 33.

La traducción de una metainstrucción a texto plano sin abuso de notación es un poco más complicada que en los otros casos. Primero se verifica que el nombre de la metainstrucción es una cadena de caracteres y que los argumentos de la metainstrucción sean una lista, después se obtiene una lista de las traducciones de cada expresión en la lista de argumentos de la metainstrucción, teniendo esta lista se concatenan comas entre cada argumento de la lista.

39 ⟨*Traducir metainstrucción a texto plano sin abuso* 39⟩≡

```
(let ([nombre     (metainstrucción-nombre     e)]
      [argumentos (metainstrucción-argumentos e)])
  (if (and (string? nombre) (list? argumentos))
      (format "~a[~a]"
              nombre
              (string-join (map expresión->texto-plano argumentos) ", "))
      (error 'metainstrucción->texto-plano
             "La metainstrucción ~a está mal formada" e)))
```

Hace referencia a `expresión->texto-plano` 33 y `metainstrucción` 5.
Este código es usado en fragmento 33.

En ocaciones será de utilidad obtener la representación externa de una expresión con las convenciones de abuso de notación utilizadas en este trabajo. La lectura de expresiones admite el abuso de notación en cualquier parte de la expresión, sin embargo, para la escritura de expresiones con abuso, habrá una sola representación externa con abuso de notación y será aquella que aproxime la máxima cantidad de abuso de notación posible.

La implementación del escritor para texto plano con abuso de notación es el procedimiento `expresión->abuso-texto-plano` el cuál tiene una forma similar a `expresión->texto-plano` sólo

que en el análisis de casos se invocan procedimientos especializados para cada clase de expresión.

40 ⟨*A texto plano con abuso* 40⟩≡

```
(define (expresión->abuso-texto-plano e)
  (cond
   [(variable? e)
    (variable->abuso-texto-plano e)]
   [(abstracción? e)
    (abstracción->abuso-texto-plano e)]
   [(aplicación? e)
    (aplicación->abuso-texto-plano e)]
   [(hueco? e)
    (hueco->abuso-texto-plano e)]
   [(metainstrucción? e)
    (metainstrucción->abuso-texto-plano e)]
   [else
    (error 'expresión->abuso-texto-plano
           "La expresión ~a no es válida" e)]))
```

Define:
expresión->abuso-texto-plano, usado en fragmentos 40, 42, 43, 40, 45, y 47.
Hace referencia a abstracción 2, abstracción->abuso-texto-plano 42, aplicación 3, aplicación->abuso-texto-plano 43, hueco 6, hueco->abuso-texto-plano 44, metainstrucción 5, metainstrucción->abuso-texto-plano 45, variable 1, y variable->abuso-texto-plano 41.
Esta definición se continua en fragmentos 41–45.
Este código es usado en fragmento 47.

Para traducir una variable a texto plano con abuso de notación, se hace uso del procedimiento `variable->texto-plano` ya que las variables tienen una sola representación externa en texto plano.

41 ⟨*A texto plano con abuso* 40⟩+≡

```
(define (variable->abuso-texto-plano e)
  (variable->texto-plano e))
```

Define:
variable->abuso-texto-plano, usado en fragmentos 40 y 42.
Hace referencia a variable 1 y variable->texto-plano 34.

La traducción de abstracciones a texto plano con abuso de notación consiste en revisar la clase de expresión que es el cuerpo, en caso que sea también una abstracción se debe iterar sobre los cuerpos de las abstracciones acumulando la representación externa de sus argumentos hasta llegar a un cuerpo que no sea abstracción. La forma de la cadena resultante debe ser λ`argumento`$_1$ `argumento`$_2$ ... `argumento`$_n$`.cuerpo`$_n$.

42 ⟨*A texto plano con abuso* 40⟩+≡

```
(define (abstracción->abuso-texto-plano e)
  (let itera ([argumentos (list (abstracción-argumento e))]
              [cuerpo     (abstracción-cuerpo e)])
    (if (abstracción? cuerpo)
        (itera (cons (abstracción-argumento cuerpo) argumentos)
               (abstracción-cuerpo cuerpo))
        (format "~a~a.~a"
                (integer->char 955)
                (string-join (map variable->abuso-texto-plano
                                  (reverse argumentos))
                             " ")
                (expresión->abuso-texto-plano cuerpo)))))
```

Define:
abstracción->abuso-texto-plano, usado en fragmentos 40 y 43.
Hace referencia a abstracción 2, expresión->abuso-texto-plano 40, variable 1, y variable->abuso-texto-plano 41.

Para traducir una aplicación a texto plano con abuso de notación primero se computa una lista de términos conformada por los operandos anidados de las aplicaciones de la forma $((M\ N)P)$ donde el primer operador que no sea aplicación es el primer elemento de la lista, es decir, se obtiene la lista de expresiones removiendo la asociación a la izquierda de la aplicación original, por ejemplo si consideramos una aplicación de la forma

$$(\ldots ((((M_1\ M_2)\ M_3)\ M_4)\ M_5)\ \ldots\ M_n)$$

la lista de expresiones seria

$$(\texttt{list}\ M_1\ M_2\ M_3\ M_4\ M_5\ \ldots\ M_n)$$

donde M_1 no es una aplicación.

Después de remover la asociación a la izquierda se determinan los elementos de la lista que deben estar encerrados entre paréntesis. El criterio utilizado en general es: si el elemento de la lista es una aplicación o una abstracción, va encerrada entre paréntesis, de lo contrario, se escribe tal cuál. Además se aplica un criterio diferente para un caso en particular: cuando el último elemento de la lista es una abstracción, no se encierra entre paréntesis, esto es para que una expresión como

$$(((M\ N)\ P)\ (\lambda x.Q))$$

pueda ser escrita como

$$M\ N\ P\ \lambda x.Q$$

la cuál es una expresión sintácticamente equivalente. Esto no se puede realizar para las abstrac-

ciones en general ya que una expresión como

$$((M\ (\lambda x.N))\ P)$$

sería considerada equivalente a

$$M\ \lambda x.N\ P$$

sin embargo, esta última expresión corresponde a

$$(M\ (\lambda x.(N\ P)))$$

al tener los paréntesis escritos de manera adecuada se concatenan las expresiones con un espacio en blanco.

El procedimiento `aplicación->abuso-texto-plano` implementa este algoritmo para el abuso de notación.

43 ⟨*A texto plano con abuso* 40⟩+≡

```
(define (aplicación->abuso-texto-plano e)
  (let itera ([operador    (aplicación-operador e)]
              [expresiones (list (aplicación-operando e))])
    (if (aplicación? operador)
        (itera (aplicación-operador operador)
               (cons (aplicación-operando operador) expresiones))
        (let itera ([expresiones  (cons operador expresiones)]
                    [traducciones null])
          (if (null? expresiones)
              (string-join (reverse traducciones) " ")
              (let ([e  (first expresiones)]
                    [es (rest  expresiones)])
                (cond
                 [(aplicación? e)
                  (itera es (cons (format "(~a)" (aplicación->abuso-texto-plano e))
                                  traducciones))]
                 [(and (abstracción? e) (not (null? es)))
                  (itera es (cons (format "(~a)" (abstracción->abuso-texto-plano e))
                                  traducciones))]
                 [else
                  (itera es (cons (expresión->abuso-texto-plano e)
                                  traducciones))])))))))
```

Define:
`aplicación->abuso-texto-plano`, usado en fragmentos 40 y 43.
Hace referencia a `abstracción` 2, `abstracción->abuso-texto-plano` 42, `aplicación` 3, y `expresión->abuso-texto-plano` 40.

La traducción a texto plano con abuso de notación para huecos es exactamente la misma que cuando no se tiene abuso de notación.

44 ⟨*A texto plano con abuso* 40⟩+≡
```
(define (hueco->abuso-texto-plano e)
  "[ ]")
```

Define:
hueco->abuso-texto-plano, usado en fragmento 40.
Hace referencia a hueco 6.

En el caso de la traducción de metainstrucciones a texto plano con abuso de notación se emplea el mismo procedimiento que en el caso donde no hay abuso de notación, con la excepción de que las llamadas recursivas a la traducción de argumentos se realizan invocando el procedimiento `expresión->abuso-texto-plano` en lugar de `expresión->texto-plano`.

45 ⟨*A texto plano con abuso* 40⟩+≡
```
(define (metainstrucción->abuso-texto-plano e)
  (let ([nombre     (metainstrucción-nombre e)]
        [argumentos (metainstrucción-argumentos e)])
    (if (and (string? nombre) (list? argumentos))
        (format "~a[~a]"
                nombre
                (string-join (map expresión->abuso-texto-plano argumentos) ", "))
        (error 'metainstrucción->abuso-text-plano
               "La metainstrucción ~a está mal formada" e))))
```

Define:
metainstrucción->abuso-texto-plano, usado en fragmento 40.
Hace referencia a expresión->abuso-texto-plano 40 y metainstrucción 5.

De expresiones a LaTeX Traducir expresiones a código interpretable por LaTeX es relativamente fácil, lo complicado es la diversidad de estilos con los que se puede escribir una expresión. Por ejemplo, el espaciado entre dos términos en una aplicación, el color de algún subtérmino, la tipografía de los símbolos o la manera en como se escriben los paréntesis. Para solventar estos problemas se reduce el problema de generar un archivo válido de LaTeX a simplemente producir una cadena de caracteres que al ser insertada en un documento válido y en un lugar donde se aplique el modo de matemáticas, el archivo seguirá siendo válido y producirá una expresión sintácticamente correcta.

Se define el parámetro `plantilla-latex` el cuál debe contener una cadena de caracteres que podrá ser utilizado de tal manera que `(format (plantilla-latex) expresión-traducida)` sea una cadena de caracteres con el contenido del documento de LaTeX válido. Para evitar complicar la implementación de este parámetro, la única condición que se verifica es que sea una cadena de caracteres y que contenga únicamente una secuencia `~a`.

46 ⟨*A LaTeX sin abuso* 46⟩≡

```
(define +plantilla-latex-inicial+
  "\
\\documentclass[preview]{standalone}\n\
\\usepackage{amsmath}\n\
\\begin{document}\n\
\\( ~a \\)\n\
\\end{document}")

(define plantilla-latex
  (make-parameter
   +plantilla-latex-inicial+
   (lambda (x)
     (if (and (string? x) (= (length (string-split x "~a" #:trim? #f)) 2))
         x
         +plantilla-latex-inicial+))))
```

Define:
plantilla-latex, usado en fragmentos 46 y 47.
Este código es usado en fragmento 47.

De expresiones a figuras funcionales de Racket

Módulo de escritor El módulo de los escritores utiliza el módulo `estructuras.rkt` y provee los procedimientos de escritura a texto plano, LaTeX y a figuras de Racket con abuso y sin abuso de notación. El archivo `escritor.rkt` contiene este módulo y tiene la siguiente forma.

47 ⟨*escritor.rkt* 47⟩≡

```
;;; -*- mode: racket; coding: utf-8 -*-
;;; Copyright (C) 2016 Eduardo Acuña Yeomans <eduardo.acye@gmail.com>
;;; Procura no editar este archivo, mejor modifica lambda.nw

#lang racket/base

(require "estructuras.rkt"
         racket/list
         racket/string)

(provide expresión->texto-plano
         expresión->abuso-texto-plano
         plantilla-latex)

;;; Traducir a texto plano sin abuso de notación
⟨A texto plano sin abuso 33⟩
```

```
;;; Traducir a texto plano con abuso de notación
⟨A texto plano con abuso 40⟩

;;; Traducir a LaTeX sin abuso de notación
⟨A LaTeX sin abuso 46⟩
```

Hace referencia a `expresión->abuso-texto-plano` 40, `expresión->texto-plano` 33, y `plantilla-latex` 46.
Fragmento raíz (no usado en este documento).

Pruebas de escritores El archivo de pruebas `escritor-pruebas.rkt` es utilizado para verificar si la funcionalidad de los escritores es la esperada. Se omite su definición en este documento pero es distribuído con el código del trabajo.

Evaluador de expresiones

Pendiente

Ciclo de Lectura Evaluación y Escritura

Pendiente

Editor estructural

Pendiente

¿Cómo hackear la λ?