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 "metainstrucciones" las cuáles 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* 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 ⟨*estructuras del lenguaje* 1⟩≡

```
(struct variable (nombre) #:transparent)
```

Define:
`variable`, usado en fragmentos 1, 4, 1, 7, 8, 23, y 25.
Esta definición se continua en fragmentos 2, 3, y 5.
Este código es usado en fragmento 9.

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

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, 2, 7, 8, 21, 2, 22, 2, y 23.

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, 8, 19, 3, 20–22, 3, y 26.

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`. Su definición es

4 ⟨*predicados del lenguaje* 4⟩≡
```
(define (término? x)
  (cond
   [(variable? x) #t]
   [(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, 6, y 8.
Hace referencia a `abstracción` 2, `aplicación` 3, y `variable` 1.
Esta definición se continua en fragmento 6.
Este código es usado en fragmento 9.

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, 6, 8, 10, y 25.

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.

6 ⟨*predicados del lenguaje* 4⟩+≡
```
(define (expresión? x)
  (or (término? x)
      (metainstrucción? x)))
```
Define:

expresión?, usado en fragmentos 8 y 23.
Hace referencia a metainstrucción 5 y término? 4.

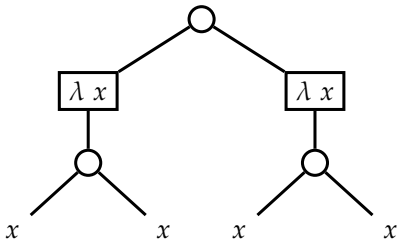
Utilizando los constructores variable, abstracción y aplicación, un término λ como

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

puede ser representado en memoria como

7 ⟨*construcción de un término* 7⟩≡
```
(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.

8 ⟨*provee las definiciones del árbol* 8⟩≡
```
(provide variable variable? variable-nombre
         abstracción abstracción? abstracción-argumento abstracción-cuerpo
         aplicación aplicación? aplicación-operador aplicación-operando
         metainstrucción metainstrucción? metainstrucción-nombre metainstrucción-argumentos
         término? expresión?)
```
Hace referencia a abstracción 2, aplicación 3, expresión? 6, metainstrucción 5, término? 4, y variable 1.
Este código es usado en fragmento 9.

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

9 ⟨*estructuras.rkt* 9⟩≡
```
;;; -*- 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

;;; Identificadores exportados
⟨provee las definiciones del árbol 8⟩

;;; 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

10 ⟨*pseudocódigo de una metainstrucción* 10⟩≡

```
(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)`.

11 ⟨*lector de expresiones* 11⟩≡

```
(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 11, 27, 11, y 28.
Hace referencia a `parsear-expresión` 12.
Esta definición se continua en fragmento 12.
Este código es usado en fragmento 28.

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.

12 ⟨*lector de expresiones* 11⟩+≡

```
(define (parsear-expresión [p (current-input-port)]
                           [c (peek-char p)]
                           [a null])
  (cond
   [(eof-object? c)
    ⟨lectura de fin de archivo 13⟩]
   [(delimitador-final? c)
    ⟨lectura de delimitador final 15⟩]
   [(char-whitespace? c)
    ⟨lectura de espacio en blanco 18⟩]
   [(char=? #\( c)
    ⟨lectura de paréntesis abierto 19⟩]
   [(or (char=? (integer->char 955) c)
        (char=? #\\ c))
    ⟨lectura de lambda 22⟩]
   [else
    ⟨lectura de variable o metainstrucción 26⟩]))
```

Define:
`parsear-expresión`, usado en fragmentos 11, 12, 18, 19, 12, 21–23, 26, y 27.
Hace referencia a `delimitador-final?` 14.

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.

13 ⟨*lectura de fin de archivo* 13⟩≡

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

Este código es usado en fragmento 12.

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

14 ⟨*procedimientos auxiliares de lectura* 14⟩≡

```
(define (delimitador-final? c)
  (or (eof-object? c)
      (char=? #\] c)
      (char=? #\) c)
      (char=? #\. c)
      (char=? #\, c)
```

```
        (char=? #\; c)))
```

Define:
delimitador-final?, usado en fragmentos 12, 14, y 17.
Esta definición se continua en fragmentos 16, 17, 20, 21, 23, 24, y 27.
Este código es usado en fragmento 28.

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.

15 ⟨*lectura de delimitador final* 15⟩≡

```
(if (null? a) (read-char p) a)
```

Este código es usado en fragmento 12.

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

16 ⟨*procedimientos auxiliares de lectura* 14⟩+≡

```
(define (delimitador-inicial? c)
  (or (char=? #\( c)
      (char=? #\[ c)
      (char=? (integer->char 955) c)
      (char=? #\\ c)))
```

Define:
delimitador-inicial?, usado en fragmentos 16 y 17.

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

17 ⟨*procedimientos auxiliares de lectura* 14⟩+≡

```
(define (delimitador? c)
  (or (delimitador-final? c)
      (delimitador-inicial? c)
      (char-whitespace? c)))
```

Define:
delimitador?, usado en fragmentos 17, 23, 24, y 28.
Hace referencia a delimitador-final? 14 y delimitador-inicial? 16.

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

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`.

18 ⟨*lectura de espacio en blanco* 18⟩≡
```
(read-char p)
(parsear-expresión p (peek-char p) a)
```
Hace referencia a `parsear-expresión` 12.
Este código es usado en fragmento 12.

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.

19 ⟨*lectura de paréntesis abierto* 19⟩≡
```
(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` 20, `parsear-expresión` 12, y `parsear-paréntesis` 21.
Este código es usado en fragmento 12.

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

20 ⟨*procedimientos auxiliares de lectura* 14⟩+≡
```
(define (aplicación/identidad a e)
  (if (null? a) e (aplicación a e)))
```

Define:
`aplicación/identidad`, usado en fragmentos 19, 20, 22, 20, y 26.
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.

21 ⟨*procedimientos auxiliares de lectura* 14⟩+≡
```
(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 21, 19, y 21.
Hace referencia a abstracción 2, aplicación 3, y parsear-expresión 12.

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.

22 ⟨*lectura de lambda* 22⟩≡

```
(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 20, parsear-abstracción 23, y parsear-expresión 12.
Este código es usado en fragmento 12.

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".

23 ⟨*procedimientos auxiliares de lectura* 14⟩+≡

```
(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 23, 22, y 23.
Hace referencia a abstracción 2, delimitador? 17, expresión? 6, parsear-constituyente 24, parsear-expresión 12, y variable 1.

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

24 ⟨*procedimientos auxiliares de lectura* 14⟩+≡
```
(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 23–25.
Hace referencia a delimitador? 17.

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.

25 ⟨*obtener variable o metainstrucción* 25⟩≡
```
(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 24, parsear-corchetes 27, y variable 1.
Este código es usado en fragmento 26.

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

26 ⟨*lectura de variable o metainstrucción* 26⟩≡
```
((lambda (e)
   (parsear-expresión p (peek-char p) (aplicación/identidad a e)))
 ⟨obtener variable o metainstrucción 25⟩)
```
Hace referencia a aplicación 3, aplicación/identidad 20, y parsear-expresión 12.
Este código es usado en fragmento 12.

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 `#\]`.

27 ⟨*procedimientos auxiliares de lectura* 14⟩+≡

```
(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 25 y 27.
Hace referencia a `leer` 11 y `parsear-expresión` 12.

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

28 ⟨*lector.rkt* 28⟩≡

```
;;; -*- 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

;;; Importar estructuras del árbol de sintaxis
(require "estructuras.rkt")

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

;;; Procedimientos principales
⟨lector de expresiones 11⟩

;;; Procedimientos auxiliares
⟨procedimientos auxiliares de lectura 14⟩
```

Hace referencia a `delimitador?` 17 y `leer` 11.
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

Pendiente

Evaluador de expresiones

Pendiente

Ciclo de Lectura Evaluación y Escritura

Pendiente

Editor estructural

Pendiente

¿Cómo hackear la λ?