UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Facultad de Ciencias

Lenguajes de Programación

MiniLisp

Proyecto 1

Presenta:
Lugo Díaz Ordaz Gretel Alexandra
Ramírez Juárez María Fernanda
Rojo Peña Manuel Ianluck

Profesor:
Manuel Soto Romero

Ayudantes:
Diego Méndez Medina
Erick Daniel Arroyo Martínez

Grupo: 7121, 2026-1

Fecha de entrega:
11 de octubre, 2025

Índice

1. Semántica Operacional **3**
1.1. Semántica Estructural (Paso pequeño) . . . 3
1.1.1. Sistema de transición . . . 3
1.1.2. Estados Finales en MiniLisp . . . 4
1.1.3. Reglas de evaluación . . . 6
1.2. Intérprete para MiniLisp . . . 10
1.2.1. Función paso pequeño en MiniLisp . . . 10
1.3. Puntos Estrictos . . . 17
1.3.1. Reglas semánticas perezosas con strict . . . 18
1.3.2. Evaluación perezosa para MiniLisp . . . 19

2. Resultados **25**
2.1. Menú interactivo . . . 25
2.2. Funciones de prueba . . . 25
2.2.1. Suma primeros n números naturales . . . 25
2.2.2. Factorial . . . 25
2.2.3. Fibonacci . . . 25
2.2.4. Función `map` para listas . . . 25
2.2.5. Función `filter` para listas . . . 25

3. Conclusiones **27**

Bibliografía **28**

Capítulo 1

Semántica Operacional

Una vez hemos establecido nuestro sintaxis del lenguaje en núcleos, lo siguiente a realizar darle el significado a estos programas. Para ello recurrimos a la semántica operacional, un enfoque que describe la dinámica de ejecución de los programas mediante un conjunto de reglas de inferencia.

Desarrollar la introducción y explicación de la formalización de la semántica y semántica operacional.

La semántica operacional...

Continuar con la definicion de semantica operacional .

Existen dos estilos principales:
Semantica Natural conocido como paso grande (Big-step)

Definir brevemente lo que es paso pequeño y quien lo definio.

Semantica Estructural conocida como paso pequeño (Small-step)

Definir brevemente lo que es paso grande y quien lo definio.

Para nuestro lenguaje nos enfocaremos en este último.

1.1. Semántica Estructural (Paso pequeño)

Se le conoce también como semántica de paso pequeño o de transición describe paso a paso la ejecución mostrando los cómputos que genera en cada paso individualmente.

Aqui ya describimos toda la historia de paso pequeño, qué significa paso pequeño y por que debemos impleemntarlo

1.1.1. Sistema de transición

Dar la definicion y explicacion de un sistema de transision y como la aplicaremos a nustro proyecto. Tambien mencionamos y describimos los estados finales pero solo eso, no los definimos para el lenguaje

Explicar lo que es la cerradura

1.1.2. Estados Finales en MiniLisp

Definimos los estados finales para el lenguaje MiniLisp -0.2cm:

$$F = \{\, Num_V(n) \mid n \in \mathbb{Z} \,\} \cup \{\, Boolean_V(b) \mid b \in \{True, False\} \,\} \cup \{\, Pair_V(f,s) \mid f,s \in F \,\} \cup \{\, Cons_V(h,t) \mid h,t \in C \,\} \cup \{\, Nil_V \mid \text{ es la lista vacía} \} \cup \{\, Closure(f,\varepsilon) \mid f \text{ es una función y } \varepsilon \text{ es un ambiente léxico} \}$$

Intuitivamente implementamos los estados finales en nuestro lenguaje como un tipo de dato en Haskell:

```haskell
module ASV where

-- ASA Values
data ASV
  = VarV String
  | NumV Int
  | BoolV Bool
  | NiV
  | PairV ASV ASV
  | ConV ASV ASV
  | Closure String ASV [(String, ASV)]
  deriving (Show, Eq)
```

Código 1.1: Tipo de dato ASV, representan los estados finales

El tipo de dato `ASV` serán con el que modelaremos los estados finales del lenguaje. Sin embargo, en nuestra implementación, es necesario agregar las demás estructuras que trabajan con estos estados finales -como lo son los operadores aritméticos o las operaciones sobre pares o listas- ya que `ASV` sigue modelando una estructura de tipo árbol la cual sigue el mismo principio que las estructuras `ASA`:

Una expresión es `ASV` si y solo si sus hijos también son `ASV`.

Por lo que debemos hacer que nuestros Para diferenciar entre los verdaderos estados finales y las demás estructuras de tipo `ASV` será que nos referimos a los finales como ***valores canónicos***.[1] Lo que nos queda en la nueva definición del tipo de dato `ASV`:

```haskell
-- ASA Values
data ASV
  = VarV String
  | NumV Int
  | BoolV Bool
  | NiV
  | AddV ASV ASV
  | SubV ASV ASV
  | MulV ASV ASV
  | DiV ASV ASV
  | SqrtV ASV
```

[1] Más adelante enfatizaremos en como los diferenciamos entre Valores `ASV` y expresiones `ASV` en Haskell.

```
  | NotV ASV
  | EqualV ASV ASV
  | LessV ASV ASV
  | GreaterV ASV ASV
  | DiffV ASV ASV
  | LeqV ASV ASV
  | GeqV ASV ASV
  | PairV ASV ASV
  | FstV ASV
  | SndV ASV
  | IfV ASV ASV ASV
  | FunV String ASV
  | AppV ASV ASV
  | ConV ASV ASV
  | HeadV ASV
  | TailV ASV
  | Closure String ASV [(String, ASV)]
 deriving (Show, Eq)
```

Código 1.2: Tipo de dato ASV completo

Ya que hemos definido el tipo de dato `ASV` veamos como convertimos nuestras estructuras `AST` a estados finales `ASV`:

```
{-- Convertimos los AST a estados finales ASV --}
toFinalState :: AST -> ASV
toFinalState (VarC x) = VarV x
toFinalState (NumC n) = NumV n
toFinalState (BoolC b) = BoolV b
toFinalState (AddC i d) = AddV (toFinalState i) (toFinalState d)
toFinalState (SubC i d) = SubV (toFinalState i) (toFinalState d)
toFinalState (MulC i d) = MulV (toFinalState i) (toFinalState d)
toFinalState (DivC i d) = DiV (toFinalState i) (toFinalState d)
toFinalState (SqrtC n) = SqrtV (toFinalState n)
toFinalState (NotC x) = NotV (toFinalState x)
toFinalState (EqualC i d) = EqualV (toFinalState i) (toFinalState d)
toFinalState (LessC i d) = LessV (toFinalState i) (toFinalState d)
toFinalState (GreaterC i d) = GreaterV (toFinalState i) (toFinalState d)
toFinalState (DiffC i d) = DiffV (toFinalState i) (toFinalState d)
toFinalState (LeqC i d) = LeqV (toFinalState i) (toFinalState d)
toFinalState (GeqC i d) = GeqV (toFinalState i) (toFinalState d)
toFinalState (PairC f s) = PairV (toFinalState f) (toFinalState s)
toFinalState (FstC p) = FstV (toFinalState p)
toFinalState (SndC p) = SndV (toFinalState p)
toFinalState (ConS f s) = ConV (toFinalState f) (toFinalState s)
toFinalState (HeadC p) = HeadV (toFinalState p)
toFinalState (TailC p) = TailV (toFinalState p)
toFinalState (IfC c t e) = IfV (toFinalState c) (toFinalState t) (
   toFinalState e)
toFinalState (FunC p b) = FunV p (toFinalState b)
toFinalState (AppC f a) = AppV (toFinalState f) (toFinalState a)
toFinalState NiL = NiV
```

Código 1.3: Función `toFinalState` que transforma los núcleos `AST` a estados finales `ASV`

La función `toFinalState` transforma cada estructura del núcleo AST en su equivalente ASV. Aunque a primera vista parezca una simple correspondencia entre constructores, su propósito es fundamental: garantizar que todas las expresiones que se evalúan dentro del intérprete sean expresadas en términos de ASV, permitiendo así que la semántica del lenguaje opere únicamente sobre estructuras homogéneas y compatibles con los valores finales del lenguaje.

1.1.3. Reglas de evaluación

Dar una introduccion de que son las reglas de evaluacion y por que son necesarias, ademas completar con la explicacion de las reglas de evaluacion. Hablar de los ambientes

- Expresiones atómicas:
  - VarV(i):

$$\frac{lookup\ i\ \varepsilon = v}{\langle VarV(i), \varepsilon\rangle \to \langle v, \varepsilon\rangle}$$

$$\frac{lookup\ i\ \varepsilon \text{ no está definido}}{\langle Varv(i), \varepsilon\rangle \to \text{Error en la ejecución de la evaluación}}$$

Aqui se puede poner la explicacion breve de lo que hace lookup

  - NumV(n):

$$\frac{}{\langle NumV(n), \varepsilon\rangle \to \langle NumV(n), \varepsilon\rangle}$$

  - BoolV(b):

$$\frac{}{\langle BoolV(b), \varepsilon\rangle \to \langle BoolV(b), \varepsilon\rangle}$$

  - NiV:

$$\frac{}{\langle NivV, \varepsilon\rangle \to \langle NiV, \varepsilon\rangle}$$

- AddV(i,d):

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon\rangle}{\langle AddV(i,d), \varepsilon\rangle \to \langle AddV(i',d), \varepsilon\rangle}$$

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon\rangle}{\langle AddV(NumV(n),d), \varepsilon\rangle \to \langle Add(NumV(n),d'), \varepsilon\rangle}$$

$$\frac{}{\langle AddV(NumV(n),NumV(m)), \varepsilon\rangle \to \langle NumV(n +_{\mathbb{Z}} m), \varepsilon\rangle}$$

- SubV(i,d):

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon\rangle}{\langle SubV(i,d), \varepsilon\rangle \to \langle SubV(i',d), \varepsilon\rangle}$$

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon\rangle}{\langle SubV(NumV(n),d), \varepsilon\rangle \to \langle SubV(NumV(n),d'), \varepsilon\rangle}$$

$$\frac{}{\langle SubV(NumV(n),NumV(m)), \varepsilon\rangle \to \langle NumV(n -_{\mathbb{Z}} m), \varepsilon\rangle}$$

- MulV(i,d):

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon\rangle}{\langle MulV(i,d), \varepsilon\rangle \to \langle MulV(i',d), \varepsilon\rangle}$$

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon\rangle}{\langle MulV(NumV(n),d), \varepsilon\rangle \to \langle MulV(NumV(n),d'), \varepsilon\rangle}$$

$$\frac{}{\langle MulV(NumV(n),NumV(m)), \varepsilon\rangle \to \langle NumV(n *_{\mathbb{Z}} m), \varepsilon\rangle}$$

- DiV(i,d):

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon\rangle}{\langle DiV(i,d), \varepsilon\rangle \to \langle DiV(i',d), \varepsilon\rangle}$$

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon\rangle}{\langle DiV(NumV(n),d), \varepsilon\rangle \to \langle DiV(NumV(n),d'), \varepsilon\rangle}$$

$$\frac{}{\langle DiV(NumV(n),NumV(0)), \varepsilon \to \text{Error en la ejecución de la evaluación}}$$

$$\frac{}{\langle DiV(NumV(n),NumV(m)), \varepsilon\rangle \to \langle NumV(n /_{\mathbb{Z}} m), \varepsilon\rangle} \quad (m \neq 0)$$

- SqrtV(n):

$$\frac{\langle n, \varepsilon\rangle \to \langle n', \varepsilon\rangle}{\langle SqrtV(n), \varepsilon\rangle \to \langle SqrtV(n'), \varepsilon\rangle}$$

$$\frac{n < 0}{\langle SqrtV(NumV(n)), \varepsilon \to \text{Error en la ejecución de la evaluación}}$$

$$\frac{}{\langle SqrtV(NumV(n)), \varepsilon\rangle \to \langle NumV(\sqrt{n}_{\mathbb{N}}), \varepsilon\rangle} \quad (n \geq 0)$$

- NotV(b):

$$\frac{\langle b, \varepsilon\rangle \to \langle n', \varepsilon\rangle}{\langle NotV(b), \varepsilon\rangle \to \langle NotV(b'), \varepsilon\rangle}$$

$$\frac{}{\langle NotV(BoolV(b)), \varepsilon\rangle \to \langle BoolV(\neg_{\mathbb{P}} b)\varepsilon\rangle}$$

- EqualV(i,d):

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon\rangle}{\langle EqualV(i,d), \varepsilon\rangle \to \langle EqualV(i',d), \varepsilon\rangle}$$

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon\rangle}{\langle EqualV(NumV(n),d), \varepsilon\rangle \to \langle EqualV(NumV(n),d'), \varepsilon\rangle}$$

$$\frac{}{\langle EqualV(NumV(n),NumV(m)), \varepsilon\rangle \to \langle BoolV(n =_{\mathbb{Z}} m), \varepsilon\rangle}$$

- LessV(i,d)

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon\rangle}{\langle LessV(i,d), \varepsilon\rangle \to \langle LessV(i',d), \varepsilon\rangle}$$

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon\rangle}{\langle LessV(NumV(n),d), \varepsilon\rangle \to \langle LessV(NumV(n),d'), \varepsilon\rangle}$$

$$\frac{}{\langle LessV(NumV(n),NumV(m)), \varepsilon\rangle \to \langle BoolV(n <_{\mathbb{Z}} m), \varepsilon\rangle}$$

- GreaterV(i,d)

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon\rangle}{\langle GreaterV(i,d), \varepsilon\rangle \to \langle GreaterV(i',d), \varepsilon\rangle}$$

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon\rangle}{\langle GreaterV(NumV(n),d), \varepsilon\rangle \to \langle GreaterV(NumV(n),d'), \varepsilon\rangle}$$

$$\frac{}{\langle GreaterV(NumV(n),NumV(m)), \varepsilon\rangle \to \langle BoolV(n >_{\mathbb{Z}} m), \varepsilon\rangle}$$

- DiffV(i,d)

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon\rangle}{\langle DiffV(i,d), \varepsilon\rangle \to \langle DiffV(i',d), \varepsilon\rangle}$$

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon\rangle}{\langle DiffV(NumV(n),d), \varepsilon\rangle \to \langle DiffV(NumV(n),d'), \varepsilon\rangle}$$

$$\frac{}{\langle DiffV(NumV(n),NumV(m)), \varepsilon\rangle \to \langle BoolV(n \neq_{\mathbb{Z}} m), \varepsilon\rangle}$$

- LeqV(i,d)

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon\rangle}{\langle LeqV(i,d), \varepsilon\rangle \to \langle LeqV(i',d), \varepsilon\rangle}$$

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon\rangle}{\langle LeqV(NumV(n),d), \varepsilon\rangle \to \langle LeqV(NumV(n),d'), \varepsilon\rangle}$$

$$\frac{}{\langle LeqV(NumV(n),NumV(m)), \varepsilon\rangle \to \langle BoolV(n \leq_{\mathbb{Z}} m), \varepsilon\rangle}$$

- GeqV(i,d)

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon\rangle}{\langle GeqV(i,d), \varepsilon\rangle \to \langle GeqV(i',d), \varepsilon\rangle}$$

$$\frac{\langle d, \varepsilon\rangle \to \langle d', \varepsilon\rangle}{\langle GeqV(NumV(n),d), \varepsilon\rangle \to \langle GeqV(NumV(n),d'), \varepsilon\rangle}$$

$$\frac{}{\langle GeqV(NumV(n),NumV(m)), \varepsilon\rangle \to \langle BoolV(n \geq_{\mathbb{Z}} m), \varepsilon\rangle}$$

Como se puede ver en las reglas para los comparadores, la regla final cuando ambas expresiones del comparador son el resultado deriva en

- PairV(f,s):

$$\frac{\langle f, \varepsilon\rangle \to \langle f', \varepsilon\rangle}{\langle PairV(f,s), \varepsilon\rangle \to \langle PairV(f',s), \varepsilon\rangle}$$

$$\frac{\langle s, \varepsilon\rangle \to \langle s', \varepsilon\rangle}{\langle PairV(v_f,s), \varepsilon\rangle \to \langle PairV(v_f,s'), \varepsilon\rangle}$$

$$\frac{v_f, v_s \text{ son valores canónicos}}{\langle PairV(v_f,v_s), \varepsilon\rangle \to \langle PairV(v_f,v_s), \varepsilon\rangle}$$

- FstV(p):

$$\frac{\langle p, \varepsilon\rangle \to \langle p', \varepsilon\rangle}{\langle FstV(p), \varepsilon\rangle \to \langle FstV(p'), \varepsilon\rangle}$$

$$\frac{v_1, v_2 \text{ son valores canónicos}}{\langle FstV(PairV(v_1, v_2)), \varepsilon\rangle \to \langle v_1, \varepsilon\rangle}$$

- SndV(p):

$$\frac{\langle p, \varepsilon\rangle \to \langle p', \varepsilon\rangle}{\langle SndV(p), \varepsilon\rangle \to \langle SndV(p'), \varepsilon\rangle}$$

$$\frac{v_1, v_2 \text{ son valores canónicos}}{\langle SndV(PairV(v_1, v_2)), \varepsilon\rangle \to \langle v_2, \varepsilon\rangle}$$

- IfV(c,t,e), IfV solo evalúa la condicional hasta llegar a un `BoolV` mas no evalúa el **then** o **else** dependiendo del resultado de la condición, solo se encarga de retornar alguno de los dos dependiendo del caso:

$$\frac{\langle c, \varepsilon\rangle \to \langle c', \varepsilon\rangle}{\langle IfV(c, t, e), \varepsilon\rangle \to \langle IfV(c', t, e), \varepsilon\rangle}$$

$$\frac{}{\langle IfV(BoolV(\#t), t, e), \varepsilon\rangle \to \langle t, \varepsilon\rangle}$$

$$\frac{}{\langle IfV(BoolV(\#f), t, e), \varepsilon\rangle \to \langle e, \varepsilon\rangle}$$

- ConV(i,d):

$$\frac{\langle i, \varepsilon\rangle \to \langle i', \varepsilon\rangle}{\langle ConV(i, d), \varepsilon\rangle \to \langle ConV(i', d), \varepsilon\rangle}$$

$$\frac{v_i \text{ es valor canónico } \langle d, \varepsilon\rangle \to \langle d', \varepsilon\rangle}{\langle ConV(v_i, d), \varepsilon\rangle \to \langle ConV(v_i, d'), \varepsilon\rangle}$$

$$\frac{v_i, v_d \text{ son valores canónicos}}{\langle ConV(v_i, v_d), \varepsilon\rangle \to \langle ConV(v_i, v_d), \varepsilon\rangle}$$

- HeadV(l):

$$\frac{\langle p, \varepsilon\rangle \to \langle p', \varepsilon\rangle}{\langle HeadV(l), \varepsilon\rangle \to \langle HeadV(l'), \varepsilon\rangle}$$

$$\frac{}{\langle HeadV(ConV(v_i, v_d)), \varepsilon\rangle \to \langle v_i, \varepsilon\rangle}$$

- TailV(l);

$$\frac{\langle l, \varepsilon\rangle \to \langle l', \varepsilon\rangle}{\langle TailV(l), \varepsilon\rangle \to \langle TailV(l'), \varepsilon\rangle}$$

$$\frac{v_d \text{ es valor pero } v_d = ConV(i, d) \;\; v_d \to v_d'}{\langle TailV(ConV(v_i, v_d)), \varepsilon\rangle \to \langle TailV(v_i, v_d'), \varepsilon\rangle}$$

$$\frac{v_d \text{ es valor pero} v_d \neq ConV(i, d)}{\langle TailV(ConV(v_i, v_d)), \varepsilon\rangle \to \langle v_d, \varepsilon\rangle}$$

- FunV(p,c):

$$\frac{}{\langle FunV(p,c),\varepsilon\rangle \to \langle Closure(FunV(p,c),\varepsilon),\varepsilon\rangle}$$

- AppV(f,a):

$$\frac{\langle f,\varepsilon\rangle \to \langle f',\varepsilon\rangle}{\langle AppV(f,a),\varepsilon\rangle \to \langle AppV(f',a),\varepsilon\rangle}$$

$$\frac{\langle a,\varepsilon\rangle \to \langle a',\varepsilon\rangle}{\langle AppV(Closure(FunV(p,c),\varepsilon'),a),\varepsilon\rangle \to \langle AppV(Closure(FunV(p,c),\varepsilon'),a'),\varepsilon\rangle}$$

$$\frac{\text{es un valor canónico}}{\langle AppV(Closure(FunV(p,c),\varepsilon'),),\varepsilon\rangle \to \langle c,\varepsilon'[p \leftarrow]\rangle}$$

Puede que sea buena idea dar un cierre a esto antes de entrar en Haskell

1.2. Intérprete para MiniLisp

Una vez definimos formalmente lo que será la ***Semántica Operacional Estrcutural*** para nuestro lenguaje podemos programar el interprete del mismo que será el encargado de aplica la evaluación al programa del usuario, más precisamente a las expreciones `ASV` que ya han depurado el programa original.

creo podemos extendernos mas aqui

1.2.1. Función paso pequeño en MiniLisp

Para nuestro lenguaje definimos la función de evaluación como sigue:

```haskell
module ASV where

import AST

{--
Definimos la representacion del ambiente de ejecucion.
Un ambiente es una lista de pares (id, valor).
--}
type Env = [(String, ASV)]

{-- ASA Values --}
data ASV
  = VarV String
  | NumV Int
  | BoolV Bool
  | NiV
  | AddV ASV ASV
  | SubV ASV ASV
  | MulV ASV ASV
  | DiV ASV ASV
  | SqrtV ASV
  | NotV ASV
```

```
  | EqualV ASV ASV
  | LessV ASV ASV
  | GreaterV ASV ASV
  | DiffV ASV ASV
  | LeqV ASV ASV
  | GeqV ASV ASV
  | PairV ASV ASV
  | FstV ASV
  | SndV ASV
  | IfV ASV ASV ASV
  | FunV String ASV
  | AppV ASV ASV
  | ConV ASV ASV
  | HeadV ASV
  | TailV ASV
  | Closure String ASV Env
 deriving (Show, Eq)
```

Código 1.4: Tipo de dato `ASV` (Estados Finales) y `ENV` (ambiente de evaluación)

Definimos tanto los estados finales como un nuevo tipo llamado Env, que representa el ambiente de evaluación. Este ambiente no es más que una emulación de la pila donde guardamos los ambientes como pares ((id, valor) como si de `["id"`$\rightarrow$ *value*`]`), donde cada identificador está asociado con el valor que tiene en ese momento de la ejecución.

Ahora definimos la función principal `eval`, para el intérprete del lenguaje:

```
module Interprete where

import ASV

{-- Funcion principal del interprete --}
{--
Evaluamos una expresion paso a paso utilizando la funcion 'pasito' hasta
    llegar a algun valor canonico.
--}
eval :: ASV -> Env -> ASV
eval asv env
  | isValue asv = asv
  | otherwise =
    let (asv', env') = pasito asv env
     in eval asv' env'

{-- Funcion que determina si una expresion es valor canonico --}
isValue :: ASV -> Bool
isValue (NumV _) = True
isValue (BoolV _) = True
isValue (Closure _ _ _) = True
isValue (PairV f s) = isValue f && isValue s
isValue (ConV f s) = isValue f && isValue s
isValue (NiV) = True
isValue _ = False
```

Código 1.5: Estados Finales

La función `eval` recibe un estado final con el ambiente de inicio el cuál es vacío al comienzo de la evaluación y devuelve un `ASV` que se al terminar será un valor canónico.
Utilizamos guardas con los que describimos que hacer. Si la expresión ya es un valor canónico (según `isValue`), no hay nada que evaluar y devolvemos el valor tal cual. En otro caso procedemos a evaluar la expresión paso a paso.
Con la función auxiliar `isValue` utilizando la casa de patrones definimos cuales estados son valores canónicos que ya establecimos y, en caso de serlo, devolvemos `True`, en otro caso no es canónico y devolvemos `False`.

Modelamos las reglas de evaluación con paso pequeño de MiniLisp -0.2cm con la función `pasito`. Esta función `pasito` recibe el estado final con su ambiente a evaluar y regresa la configuración resultante:

```
{--  Funcion pasito que implementa la semantica operacional estructural
    del lenguaje --}
{--
'Avanza' un solo paso en la evaluacion de una expresion, devolviendo el
    resultado y el ambiente actualizado.
--}
pasito :: ASV -> Env -> (ASV, Env)
--Valores
pasito (VarV i) env  = (mirarriba i env, env)
pasito (NumV n) env  = (NumV n, env)
pasito (BoolV b) env = (BoolV b, env)
pasito (NiV) env     = (NiV, env)


{-- Funcion mirarriba (lookup) para la busqueda de variables en el
    ambiente --}
mirarriba :: String -> Env -> ASV
mirarriba i [] = error ("Var '" ++ i ++ "' no definida")
mirarriba i ((j, v):e)
  | i == j = v
  | otherwise = mirarriba i e
```

Código 1.6: Función `pasito` para valores, casos base

Como se puede ver, `pasito` utiliza la casa de patrones de acuerdo con las reglas que establecimos previamente. `NumV n`, `BoolV b` y `NiV` solo devuelven la misma expresión con el mismo ambiente recibido.

Tenemos la mención especial de que para las variables `VarV` utilizamos `lookup`, que en nuestra implementación, con un arrebato increíble de originalidad lo nombramos como `mirarriba`; para buscar las varibales en el ambiente dado. La función implementa la búsqueda de una variable en el ambiente, toma el nombre de una variable y un ambiente `Env` y devuelve el `ASV` asociado.
Si el ambiente es vacío o llegamos al final de este y no encontramos la variable, lanzamos un error con un mensaje indicando que la variable no está definida, deteniedno la ejecución de todo el intérprete. En el caso recursivo, comprobamos que el identificador en el ambiente sea el mismo que el ambiente que se está buscando, en caso de que coincidan devolvemos el valor

v asociado ya que hemos encontrado la variable. En otro caso, si no coinciden, continuamos la búsqueda recursivamente en la cola. Recorremos el ambiente hasta encontrar la variable o agotar la "*pila*".

```haskell
pasito :: ASV -> Env -> (ASV, Env)
-- Operadores
pasito (AddV (NumV n) (NumV m)) env = (NumV (n + m), env)
pasito (AddV (NumV n) d) env        = let (d', env') = pasito d env
                                      in (AddV (NumV n) d', env')
pasito (AddV i d) env               = let (i', env') = pasito i env
                                      in (AddV i' d, env')
pasito (SubV (NumV n) (NumV m)) env = (NumV (n - m), env)
pasito (SubV (NumV n) d) env        = let (d', env') = pasito d env
                                      in (SubV (NumV n) d', env')
pasito (SubV i d) env               = let (i', env') = pasito i env
                                      in (SubV i' d, env')
pasito (MulV (NumV n) (NumV m)) env = (NumV (n * m), env)
pasito (MulV (NumV n) d) env        = let (d', env') = pasito d env
                                      in (MulV (NumV n) d', env')
pasito (MulV i d) env               = let (i', env') = pasito i env
                                      in (MulV i' d, env')
pasito (DiV (NumV n) (NumV m)) env
  | m == 0 = error ("No se puede dividir entre 0")
  | otherwise = (NumV (div n m), env)
pasito (DiV (NumV n) d) env = let (d', env') = pasito d env
                              in (DiV (NumV n) d', env')
pasito (DiV i d) env        = let (i', env') = pasito i env
                              in (DiV i' d, env')
pasito (SqrtV (NumV n)) env
  | n < 0 = error ("No se puede obtener la raiz de un numero negativo")
  | otherwise = (NumV (integerSquareRoot n), env)
pasito (SqrtV n) env         = let (n', env') = pasito n env
                               in (SqrtV n', env')

{-- Funcion auxiliar para SqrtV para calcular la raiz cuadrada de un
    numero entero --}
integerSquareRoot :: Int -> Int
integerSquareRoot n = floor (sqrt (fromIntegral n :: Double))
```

Código 1.7: Función `pasito` para operadores

Para las siguientes reglas de evaluación definiremos primero la tercer regla de cada expresión donde los arguementos de estas ya son valores, ya que al usar *pattern matching* debemos implementarlo de este modo pues de otra forma caeríamos siempre en la primer regla donde solo evaluamos un argumento, lo que derivaría en un bucle infinito. Así, la regla para los operadores utiliza las reglas donde ambos argumentos son ya números para terminar la evaluación, donde usamos los operadores del lenguaje anfitrión respectivos a la expresión.
Los casos especiales son para la división y la raíz cuadrada, donde, una vez llegamos a un `NumV`, comprobamos que no sea cero, para la división, o que no sea negativo, para la raíz. En el caso de `SqrtV`, una vez comprobamos que el número es válido, nos apoyamos de la función auxiliar `integerSquareRoot` que convierte el entero a `Double`, calcula la raíz cuadrada y la trunca hacia abajo, devolviendo un `Int`.

De manera similar implementamos las reglas para los comparadores:

```
pasito :: ASV -> Env -> (ASV, Env)
--Not
pasito (NotV (BoolV b)) env = (BoolV (not b), env)
pasito (NotV e) env         = let (e', env') = pasito e env
                              in (NotV e', env')
--Comparadores
pasito (EqualV (NumV n) (NumV m)) env   = (BoolV (n == m), env)
pasito (EqualV (NumV n) d) env          = let (d', env') = pasito d env
                                          in (EqualV (NumV n) d', env')
pasito (EqualV i d) env                 = let (i', env') = pasito i env
                                          in (EqualV i' d, env')
pasito (LessV (NumV n) (NumV m)) env    = (BoolV (n < m), env)
pasito (LessV (NumV n) d) env           = let (d', env') = pasito d env
                                          in (LessV (NumV n) d', env')
pasito (LessV i d) env                  = let (i', env') = pasito i env
                                          in (LessV i' d, env')
pasito (GreaterV (NumV n) (NumV m)) env = (BoolV (n > m), env)
pasito (GreaterV (NumV n) d) env        = let (d', env') = pasito d env
                                          in (GreaterV (NumV n) d', env')
pasito (GreaterV i d) env               = let (i', env') = pasito i env
                                          in (GreaterV i' d, env')
pasito (DiffV (NumV n) (NumV m)) env    = (BoolV (n /= m), env)
pasito (DiffV (NumV n) d) env           = let (d', env') = pasito d env
                                          in (DiffV (NumV n) d', env')
pasito (DiffV i d) env                  = let (i', env') = pasito i env
                                          in (DiffV i' d, env')
pasito (LeqV (NumV n) (NumV m)) env     = (BoolV (n <= m), env)
pasito (LeqV (NumV n) d) env            = let (d', env') = pasito d env
                                          in (LeqV (NumV n) d', env')
pasito (LeqV i d) env                   = let (i', env') = pasito i env
                                          in (LeqV i' d, env')
pasito (GeqV (NumV n) (NumV m)) env     = (BoolV (n <= m), env)
pasito (GeqV (NumV n) d) env            = let (d', env') = pasito d env
                                          in (GeqV (NumV n) d', env')
pasito (GeqV i d) env                   = let (i', env') = pasito i env
                                          in (GeqV i' d, env')
```

Código 1.8: Función `pasito` para comparadores

En el caso de `Not`, una vez que el argumento es `BoolV` retornamos la negación en Haskell de este `BoolV` como un nuevo `BoolV`. De la misma manera como fue con los operadores aritméticos, cuando ambos argumentos ya son números con los cuales comparar, regresamos la comparación como valor booleano`BoolV` con el resultado evaluado con los operadores de comparación en Haskell.

```
pasito :: ASV -> Env -> (ASV, Env)
--Pares
pasito (PairV f s) env
  | isValue (PairV f s) = (PairV f s, env)
  | isValue f = let (s', env') = pasito s env
                in (PairV f s', env')
```

```haskell
  | otherwise = let (f', env') = pasito f env
                in (PairV f' s, env')
pasito (FstV (PairV f s)) env
  | isValue f && isValue s = (f, env)
pasito (FstV p) env = let (p', env') = pasito p env
                      in (FstV p', env')
pasito (SndV (PairV f s)) env
  | isValue f && isValue s = (s, env)
pasito (SndV p) env = let (p', env') = pasito p env
                      in (SndV p', env')
---Cons
pasito (ConV f s) env
  | isValue (ConV f s) = (ConV f s, env)
  | isValue f = let (s', env') = pasito s env
                in (ConV f s', env')
  | otherwise = let (f', env') = pasito f env
                in (ConV f' s, env')
pasito (HeadV (ConV f s)) env
  | isValue f && isValue s = (f, env)
pasito (HeadV p) env =
  let (p', env') = pasito p env
  in (HeadV p', env')
pasito (TailV (ConV f s)) env
  | isValue f && isValue s && not (isConV s) = (s, env)
  | otherwise = let (s', env') = pasito s env
                in (TailV s', env')
pasito (TailV p) env =
  let (p', env') = pasito p env
  in (TailV p', env')


{-- Funcion que nos dice si la estructura sigue siendo un ecadenamiento de
    ConV --}
isConV :: ASV -> Bool
isConV (ConV _ _) = True
isConV _ = False
```

Código 1.9: Función `pasito` para comparadores

Para el caso de `PairV` y `ConV`, en ambos, comprobamos que sean valores canónicos y aunque suene redundante, con la función `isValue`, ambos son valores si y sólo si sus dos argumentos también son valores. En otro caso continuamos evaluando ambas expresiones, primero con el primer argumento hasta llegar a un valor para poder seguir con la evalución del segundo.
Para la evaluación de los operadores sobre pares, con `FstV`, como utilizamos evaluación glotona, debemos llegar a que el par recibido tenga ambas expresiones como valor, para así devolver el primer argumento del par. Lo mismo con `SndV` que, hasta tener los argumentso f y s como valores, devolvemos s, el segundo.

Para el caso de `HeadV` con `ConV` es análogo a `FstV` con los pares. Sin embargo, `TailV` no solo necesita que ambos argumentos ya seal valores, sino que el segundo valor no sea de tipo `ConV`, pues esto significa que hay más expresiones en el encadenamiento de `ConV`, por lo que

necesitamos seguir con este segundo argumento hasta llegar a un valor canónico dinstinto de `ConV` -justo como en las reglas de evaluación para esta expresión-.
Para lograr esto, no apoyamos en la función `isConV`, que nos ayuda a determinar que la expresión dada sea otro `ConV` o si es otro valor canónico.

Para las reglas de la condicional `IfV` su implementación es más sencilla pues basta que, al llegar a un valor `BoolV` de la condición devolvemos la configuración respectiva: `(t, env)` en el caso de que la condición es verdad, y `(e, env)` en otro caso. De otro modo, continuamos evaluando la condición.

```
pasito :: ASV -> Env -> (ASV, Env)
--If
pasito (IfV (BoolV True) t e) env  = (t, env)
pasito (IfV (BoolV False) t e) env = (e, env)
pasito (IfV c t e) env             = let (c', env') = pasito c env
                                     in (IfV c' t e, env')
```

Código 1.10: Función `pasito` para comparadores

Por último tenemos las funciones y `FunV` las aplicaciones de funciones `AppV`, que quedan como sigue:

```
pasito :: ASV -> Env -> (ASV, Env)
--Funciones
pasito (FunV p c) env = (Closure p c env, env)
--Aplicacion de funciones
pasito (AppV (Closure p c env') a) env
  | isValue a = (c, (p, a):env')
  | otherwise =
    let (a', env'') = pasito a env
    in (AppV (Closure p c env') a', env'')
pasito (AppV f a) env = let (f', env') = pasito f env
                        in (AppV f' a, env')
```

Código 1.11: Función `pasito` para comparadores

La evaluación de las funciones es simple, la función `FunV` se convierte en una cerradura con el mismo ambiente de la configuración. Para `AppV`, siguiendo las reglas de evaluación para esta expresión: si se tiene una cerradura con un argumento a aplicar, comprobamos que este argumento sea ya un valor canónico, de ser así, retornamos la configuración con la aplicación extendiendo el ambiente. En otro caso continuamos evaluacndo el argumento. De no ser una cerradura, entonces evaluamos esta función.

Así, terminamos la evaluación glotona con alcance estático usando paso pequeño con la función `pasito` en nuestro intérprete.

Sin embargo, esta implementación no nos sirve para la aplicación de funciones derivada de `LetRec`, pues veamos el resultado de evaluar `LetRec`:

```
[MiniLisp]> (letrec (sum (lambda (n) (if0 n 0 (+ n (sum (- n 1)))))) (sum 3))
[Error]: Var 'Z' no definida
```

```
CallStack (from HasCallStack):
  error, called at ./Interprete.hs:154:18 in main:Interprete
```

Y es debido a que, como se menciono en en el capítulo de **Azúcar Sintáctica**, necesitamos definir un *Combinador* para darle recursión. Más concretamente, usamos el combinador **Z**:

$$(\lambda f.((\lambda x.(f(\lambda v.((xx)v))))(\lambda x.(f(\lambda v.((xx)v))))))$$

creo se puede mejorar la explicacion anterior

Y este combinador debemos agregarlo al ambiente de evaluación para que en el interpretre lo utilice, y asu vez, el ambiente debe estar evaluado:

$$[\mathtt{Z} \rightarrow (\lambda f.((\lambda x.(f(\lambda v.((xx)v))))(\lambda x.(f(\lambda v.((xx)v))))))]$$

No obstante, esto también presenta un error a la hora de ser evaluado:

```
[MiniLisp]> (letrec (sum (lambda (n) (if0 n 0 (+ n (sum (- n 1)))))) (sum 3))
[Error]: Var 'v' no definida
CallStack (from HasCallStack):
  error, called at ./Interprete.hs:154:18 in main:Interprete
```

Debido a que hasta el momento estamos utilizamos evaluación glotona con alcance estático, debemos de implementar una manera de que no suceda esto. Y es con...

Extender aqui la explicacion de por que falla al evaluar el combinador como el uso de evaluacion glotona

1.3. Puntos Estrictos

Para evitar los de variables no definidas al utilizar recursion y combinadores, debemos volver a MiniLisp perezo, es decir, debemos reeplantear la estrategia de evaluacion de MiniLisp de glotón a perezoso.

Al usar evaluación perezosa, existen ciertos puntos dentro de la implementación en los que no puede postergarse la evaluación.

Explicar que es el alcance perezoso y lo que son puntos estrictos.

Para ello tenemos que definir un forma que nos permitia aplicar los *puntos estrictos* de nuestro lenguaje. Este forma la modelaremos mediante una función **strict**, que reduce la expresión hasta llegar a un **valor canónico** del lenguaje.

$$\mathbf{strict}(\langle e, \varepsilon\rangle) = \begin{cases} e, & \text{si } e \text{ es valor,} \\ \langle e', \varepsilon\rangle, & \text{si } \langle e, \varepsilon\rangle \rightarrow \langle e', \varepsilon'\rangle. \end{cases}$$

Para aplicar este mecanismo, sí o sí necesitamos nuestras reglas semánticas de modo que apliquen los *puntos estrictos* correspondientes usando **strict**. El resto de reglas se mantienen sin cambios, pero el cambio radica en que: aquellas expresiones donde detectamos la presencia de *puntos estrictos* y recibian cerraduras de expresión $\langle e, \varepsilon \rangle$, es donde llamamos **strict**.

1.3.1. Reglas semánticas perezosas con strict

- AddV(i,d):
- SubV(i,d):
- MulV(i,d):
- DiV(i,d):
- SqrtV(n):
- NotV(b):
- EqualV(i,d):
- LessV(i,d):
- GreaterV(i,d):
- DiffV(i,d):
- LeqV(i,d):
- GeqV(i,d):
- PairV(f,s):
- FstV(p):
- SndV(p):
- IfV(c,t,e):
- ConV(i,d):
- HeadV(l):
- TailV(l);
- FunV(p,c):
- AppV(f,a):

Consultar la pagina 13-14 del ultimo pdf del profesor para ver contexto

Una vez definido todo lo anterior, veamos su implementación en Haskell y cómo se modifica el interprete de MiniLisp -0.2cm.

1.3.2. Evaluación perezosa para MiniLisp

Como se mostró, al cambiar la estrategia de evaluación de nuestro intérprete, debemos hacer muchos cambios en su implementación en Haskell.

Con este nuevo enfoque, nuestro tipo de dato `ASV` ya es más reducido donde solo están los valores canónicos:

```haskell
module ASV where

import AST

{--
Definimos la representacion del ambiente de ejecucion.
Un ambiente es una lista de pares (id, valor).
--}
type Env = [(String, ASV)]

{-- ASA Values --}
data ASV
  = NumV Int
  | BoolV Bool
  | NiV
  | PairV ASV ASV
  | ConV ASV ASV
  | ClosureF String AST Env
  | ExprV AST Env
  deriving (Show, Eq)
```

Código 1.12: Tipo de dato `ASV` con valores canónicos

A diferencia de la implementación anterior, en este nuevo interprete ya no tenemos que usar la función `toFinalState` para convertir los **ASA** en estados finales, ya que los únicos estados finales son los valores canónicos (Ambas definiciones significan lo mismo pero recordemos que en la primer implementación hicimos el abuso de notación para marcar la diferencia entre ambos, de eso se ecargará el nuevo intérprete, de convertir las expresion `AST` a estados finales (valores canónicos).

Definimos la función de evaluación perezosa con puntos estrictos como sigue:

```haskell
module EvalStrict where

import AST
import ASV
import Interprete

{-- Funcion de evaluacion perezosa --}
evalS :: AST -> Env -> ASV
--Valores
evalS (VarC i) env = lookupS i env
evalS (NumC n) _   = (NumV n)
evalS (BoolC b) _  = (BoolV b)
evalS NiL _        = NiV
--Operadores aritmeticos
```

```
evalS (AddC i d) env =
  let i' = strict (evalS i env)
      d' = strict (evalS d env)
   in NumV (numN i' + numN d')
evalS (SubC i d) env =
  let i' = strict (evalS i env)
      d' = strict (evalS d env)
   in NumV (numN i' - numN d')
evalS (MulC i d) env =
  let i' = strict (evalS i env)
      d' = strict (evalS d env)
   in NumV (numN i' * numN d')
evalS (DivC i d) env =
  let n = numN (strict (evalS i env))
      m = numN (strict (evalS d env))
   in if m == 0
        then error "No se puede dividir entre 0"
        else NumV (div n m)
evalS (SqrtC n) env =
  let n' = numN (strict (evalS n env))
   in if n' < 0
        then error "No se puede obtener la raiz de un numero negativo"
        else NumV (integerSquareRoot n')
--Not
evalS (NotC e) env =
  let e' = strict (evalS e env)
   in BoolV (not (boolN e'))
--Comparadores
evalS (EqualC i d) env =
  let i' = strict (evalS i env)
      d' = strict (evalS d env)
   in BoolV (numN i' == numN d')
evalS (LessC i d) env =
  let i' = strict (evalS i env)
      d' = strict (evalS d env)
   in BoolV (numN i' < numN d')
evalS (GreaterC i d) env =
  let i' = strict (evalS i env)
      d' = strict (evalS d env)
   in BoolV (numN i' > numN d')
evalS (DiffC i d) env =
  let i' = strict (evalS i env)
      d' = strict (evalS d env)
   in BoolV (numN i' /= numN d')
evalS (LeqC i d) env =
  let i' = strict (evalS i env)
      d' = strict (evalS d env)
   in BoolV (numN i' <= numN d')
evalS (GeqC i d) env =
  let i' = strict (evalS i env)
      d' = strict (evalS d env)
   in BoolV (numN i' >= numN d')
--Pares
evalS (PairC i d) env =
```

```
  let i' = strict (evalS i env)
      d' = strict (evalS d env)
   in PairV i' d'
evalS (FstC p) env =
  case strict (evalS p env) of
    PairV f _ -> f
    _         -> error "Fst espera un par"
evalS (SndC p) env =
  case strict (evalS p env) of
    PairV _ s -> s
    _         -> error "Snd espera un par"
--Cons
evalS (ConS i d) env =
  let i' = strict (evalS i env)
      d' = strict (evalS d env)
   in ConV i' d'
evalS (HeadC p) env =
  case strict (evalS p env) of
    ConV h _ -> h
    _        -> error "Head espera una lista"
evalS (TailC p) env =
  case strict (evalS p env) of
    ConV _ t ->
      let t' = strict t
       in if not (isConV t')
          then t'
          else tailDeep t'
    _        -> error "Tail espera una lista"
--If
evalS (IfC c t e) env =
  let cond = boolN (strict (evalS c env))
   in if cond then evalS t env else evalS e env
--Funciones
evalS (FunC p c) env = ClosureF p c env
--Aplicacion de funciones
evalS (AppC f a) env =
  let f' = evalS f env
      funV = strict f'
   in evalS (closureC funV) (((closureP funV), ExprV a env) : (closureE
      funV))

{-- Funcion strict para forzar la evaluacion de los puntos estrictos --}
strict :: ASV -> ASV
strict (NumV n) = NumV n
strict (BoolV b) = BoolV b
strict (PairV f s) = PairV (strict f) (strict s)
strict (ConV i d) = ConV (strict i) (strict d)
strict (ExprV a e) = strict (evalS a e)
strict (NiV) = NiV
strict (ClosureF p c e) = ClosureF p c e

{-- Funcion auxiliar para devolver el numero de NumV--}
numN :: ASV -> Int
numN (NumV n) = n
```

```
122
123 {-- Funcion auxiliar para devolver el booleano de BoolV --}
124 boolN :: ASV -> Bool
125 boolN (BoolV b) = b
126 boolN _ = False
127
128 {-- Funcion auxiliar para devoler el parametro de la cerradura --}
129 closureP :: ASV -> String
130 closureP (ClosureF p _ _) = p
131
132 {-- Funcion auxiliar para devoler el cuerpo de la cerradura --}
133 closureC :: ASV -> AST
134 closureC (ClosureF _ c _) = c
135
136 {-- Funcion auxiliar para devoler el ambiente de la cerradura --}
137 closureE :: ASV -> Env
138 closureE (ClosureF _ _ e) = e
139
140 {-- Funcion auxiliar para encontrar el ultimo elemento canonico de los
       ConV anidados--}
141 tailDeep :: ASV -> ASV
142 tailDeep (ConV _ rest) =
143   let rest' = strict rest
144   in if not (isConV rest')
145      then rest'
146      else tailDeep rest'
147 tailDeep v = v
```

Código 1.13: Evaluación perezosa con **strict** para MiniLisp -0.2cm

La mayor diferencia a `eval` es que la función `evalS` ya no utiliza la evaluación de paso pequeño, sino que implementa una evaluación directa (***big-step***). Ahora utilizamos funciones auxiliares (`strict`, `numN`, `boolN`, etc.) para obtener valores "*forzados*".
Además de este cambio, lo más notorio es `TailC` y la aplicación de funciones. La primer expresión ahora utiliza una función auxiliar `tailDeep` para seguir buscando el último elemento en la anidación de `ConS`, ya que usar `evalS` nos da una inconsistencia de tipos, pues al evaluar por primera vez el par, este se transforma en un `ASV` siendo que `evalS` recibe `AST`.
Para el caso de `AppC`:

1. Evalúa f en el ambiente actual, esto debería devolver una cerradura.
2. Fuerza f' con `strict`, asegurándose de que sea una cerradura concreta.
3. Extrae información de la cerradura:
   - `closureC funV`: el cuerpo de la función.
   - `closureP funV`: el nombre del parámetro formal.
   - `closureE funV`: el ambiente donde la función fue creada.
4. Crea un nuevo ambiente donde:
   - El parámetro formal (`closureP funV`) se asocia a la expresión real (`ExprV a env`);

- Se añade al ambiente original de la cerradura (`closureE funV`).

5. Evalúa el cuerpo de la función (`closureC funV`) en ese nuevo ambiente extendido.

De este modo tenemos una evaluación adecuada para introducir recursión con `LetRec` en nuestro lenguaje MiniLisp -0.2cm, pues como mencionamos, necesitamos meter al ambiente inicial la evaluación de nuestro combinador **Z**, el cuál se mostró que presentaba errores con la evaluación glotona.

Aqui podemos poner un cierre del interprete, recursion y evaluacion glotona

Capítulo 2

Resultados

Bien, una vez hemos visto toda la teoría de nuestro MiniLisp y además de que hemos mostrado el código que lo implementa en Haskell, veamos como funciona:

2.1. Menú interactivo

2.2. Funciones de prueba

2.2.1. Suma primeros n números naturales

2.2.2. Factorial

2.2.3. Fibonacci

2.2.4. Función `map` para listas

2.2.5. Función `filter` para listas

Capítulo 3

Conclusiones

Bibliografía

[1] https://weblibrary.mila.edu.my/upload/ebook/engineering/2017_Book_FoundationsOfProgr

[2] Aho, A. V., Lam, S. M., Sethi, R., & Ullman, J. D. Compilers: Principles, Techniques, and Tools. [Second Edition]. 2007.

[3] Disponible en: https://lambdasspace.github.io/LDP/notas/ldp_n04.pdf

[4] Disponible en: https://lambdasspace.github.io/LDP/notas/ldp_n05.pdf

[5] Documentación Haskell. Disponible en: https://www.haskell.org

[6] Documentación Alex(Haskell) The Alex Lexer Generator for Haskell Programming in Haskell (Graham Hutton, 2nd Edition). Sección sobre parsers y lexers. Disponible en: https://www.haskell.org/alex/

[7] Marlow, S., Gill, A. (2009). Happy. Disponible en: https://www.haskell.org/happy/

[8] Disponible en: https://lambdasspace.github.io/LDP/notas/ldp_n08.pdf

[9] Disponible en: https://lambdasspace.github.io/LDP/notas/ldp_n09.pdf

[10] Disponible en: https://lambdasspace.github.io/LDP/notas/ldp_n10.pdf

[11] Disponible en: https://docs.racket-lang.org/reference/let.html

[12] Disponible en: https://www.lispworks.com/documentation/HyperSpec/Body/s_let_l.htm