Solucionario del curso de Lenguajes de Programacion 2019-1

Emiliano Galeana Araujo
Material elaborado bajo el proyecto PAPIME PE 102117

7 de enero de 2019

Índice general

Parte I

Ejercicios Semanales

Ejercicio Semanal 2

- Definir un juicio `ProgPF` que formalice:
  Un programa en `PostFix` es una secuencia parentizada compuesta de:
  - La palabra reservada `postfix`.
  - Un número natural n que indica el número de argumentos de entrada.
  - Una sucesión posiblemente vacía de comandos.

$$\frac{\text{n N} \quad \text{a Arg}}{\text{(postfix n a) ProgPF}}\ PPF$$

$$\frac{\text{a SeqC}}{\text{a Arg}}\ arg1$$

$$\frac{\text{a SeqE}}{\text{a Arg}}\ arg2$$

$$\frac{\text{a Arg} \quad \text{b Arg}}{\text{a b Arg}}\ arg3$$

- Muestre mediante una derivación con su definición y las reglas definidas previamente en la nota de clase 3 que:
  `(postfix 2 nget -7 pop (exec -6 swap)) ProgPF`

1. ϵ secC
2. swap com
3. swap ϵ seqC(1,2)

4. -6 com
5. -6 swap ϵ seqC(4,3)
6. exec com
7. exec -6 swap ϵ seqC(6,5)
8. (exec -6 swap) seqE
9. pop com
10. pop ϵ seqC(9)
11. pop (exec -6 swap) arg(10,8)
12. -7 com
13. -7 ϵ seqC(12)
14. -7 pop (exec -6 swap) arg(13, 11)
15. nget com
16. nget ϵ seqC(15)
17. nget -7 pop (exec -6 swap) arg (15, 13)
18. 2 $\mathbb{N}$
19. 2 nget -7 pop (exec -6 swap) PPF (18)
20. (postfix 2 nget -7 pop (exec -6 swap)) ProgPF 19

Ejercicio Semanal 3

Considere la siguiente gramática G:
e ::= 0 |2 |e + e |e * e
donde + es la suma y * es la multiplicación convencional.

- Dar una definición inductiva de la gramática G mediante juicios w E cuya especificación es w E si y solo si w se genera con la gramática G.

$$\frac{}{0\ E}$$

$$\frac{}{2\ E}$$

$$\frac{e_1\ E \quad e_2\ E}{e_1 + e_2\ E}$$

$$\frac{e_1\ E \quad e_2\ E}{e_1 * e_2\ E}$$

- Demuestre que todas las expresiones generadas por G son números pares, usando inducción estructural sobre el juicio w E. Debe de enunciar también las reglas que se utilizan para la definición de par.
  P.D: Las expresiones generadas por G son números pares.
  Def: Un número par es un número de la forma *2n* con n $\in \mathbb{N}$.
  En juicios sería: $\frac{n \in \mathbb{N}}{2n\ par}$
  Demostración (Inducción estructural)

- base:
  - 0 ✓ ya que es de la forma 2(0) = 0.
  - 2 ✓ ya que es de la forma 2(1) = 2.
- hipótesis: suponemos que e_1 E y e_2 E *i.e.* e_1 = 2m y e_2 = 2n con m,n ∈ $\mathbb{N}$.
- Paso inductivo
  - e_1 + e_2 E. e_1 + e_2 = 2m + 2n = 2(m + n) = e_3 E.
  - e_1 * e_2 E. e_1 * e_2 = 2m * 2n = 2(m * n) = e_3' E.

- Defina la sintáxis abstracta para este lenguanje mediante el juicio t *asa*.

$$\frac{}{[0] asa}$$

$$\frac{}{[2] asa}$$

$$\frac{t_1 \ asa \quad t_2 \ asa}{suma(t_1, t_2) \ asa}$$

$$\frac{t_1 \ asa \quad t_2 \ asa}{mul(t_1, t_2) \ asa}$$

Ejercicio Semanal 4

- Dada la siguiente expresión e:

```
let x = let y = 2 in 3 * y end in
    let w = 10 in
      ley y = w - x in y * x end + 2
    end
end
```

  - Llenar la siguiente tabla con base en e. El orden de las expresiones let es, por supuesto, el orden de lectura como texto.

let	variable ligada	Exp. a ligar	alcance
1	x	let y = 2 in 3 * y end	let w = 10 in let y = w - x in y * x end + 2 end
2	y	2	3 * y
3	w	10	let y = w - x in y * x end + 2
4	y	w - x	y * x

  - Encontrar la representación en la sintáxis abstracta de orden superior de e. Está permitido usar la siuntáxis concreta para operaciones y números.
    let(let(num[2],y.prod(num[3],y)),x.let(num[10],w.sum(num[2],let(res(w, x), y.prod(y, x)))))
  - Encontrar una expresión e' que sea α-equivalente a e y en donde todas las variables ligadas tengan distinto nombre.

```
let a = let b = 2 in 3 * b end in
    let c = 10 in
      ley d = c - a in d * a end + 2
```

```
    end
end
```

- ¿Cuál es el valor de la expresión e? Explicar.
  26, primero x = 3 * y (que vale 2), entonces x = 6. Eso lo pasamos a w = 10 y lo pasamos a y = 4. Todo esto, al llegar a la multiplicación regresa 24, por último, sumamos 2.

- Realiza la siguiente sustitución mostrando la respuesta paso a paso:
  `(let x = y + 4 in (let z = x * 2 in y + 1 end) * y end) [y := x * 2]`
  Primero hacemos una α-equivalencia.
  `(let a = y + 4 in (let b = a * 2 in y + 1 end) * y end) [y := x * 2]`
  Realizamos la sustitución.
  `(let a = (x * 2) + 4 in (let b = a * 2 in (x * 2) + 1 end) * (x * 2) end)`

Ejercicio Semanal 5

Extendemos el lenguaje EAB con un tipo primitivo de números reales *positivos* como sigue:

- Tipo: `PFloat`
- Expresiones en sintáxis concreta:

  e ::= ... |r |e ÷ e |fl e

  donde r es cualquier real positivo, ÷ es el operador de división y fl denota a la función piso. Estas operaciones sólo tienen sentido para reales positivos.

- Definir la sintáxis abstracta para esta extensión.

$$\frac{n \text{ número real positivo }^{1}}{numr[n]\ asa}$$

$$\frac{t_1\ asa}{fl(t_1)\ asa}$$

$$\frac{t_1\ asa \quad t_2\ asa}{div(t_1, t_2)\ asa}$$

- Definir la semántica operacional para esta extensión. Utilizando la estrategia de *derecha a izquierda* y llamada por valor. Se debe definir previamente qué expresiones son valores. *Sugerencia:* puede suponer

[1] Menos el 0

definidas las operaciones / y *floor* para división y piso de valores reales positivos, respectivamente.

Valores:

$$\frac{}{\text{numr[r] final}}\ \mathit{freal}$$

Mas las ya definidas en EAB.

Juicios[2,3]:

$$\frac{}{\text{suma(num[n], num[m])} \to \text{num[n + m]}}\ \mathit{esumaf}$$

$$\frac{t_2 \to t_2'}{\text{suma}(t_1, t_2) \to \text{suma}(t_1, t_2')}\ \mathit{esumai}$$

$$\frac{t_1 \to t_1'}{\text{suma}(t_1, \text{num[n]}) \to \text{suma}(t_1', \text{num[n]})}\ \mathit{esumad}$$

$$\frac{}{\text{prod(num[n], num[m])} \to \text{num[n * m]}}\ \mathit{eprodf}$$

$$\frac{t_2 \to t_2'}{\text{prod}(t_1, t_2) \to \text{prod}(t_1, t_2')}\ \mathit{eprodi}$$

$$\frac{t_1 \to t_1'}{\text{prod}(t_1, \text{num[n]}) \to \text{prod}(t_1', \text{num[n]})}\ \mathit{eprodd}$$

$$\frac{}{\text{div(numr[r], numr[s])} \to \text{numr[r / s]}}\ \mathit{edivf}$$

$$\frac{t_2 \to t_2'}{\text{div}(t_1, t_2) \to \text{div}(t_1, t_2')}\ \mathit{edivi}$$

$$\frac{t_1 \to t_1'}{\text{div}(t_1, \text{numr[r]}) \to \text{div}(t_1', \text{numr[r]})}\ \mathit{edivd}$$

$$\frac{}{\text{fl(numr[r])} \to \text{num[floor(r)]}}\ \mathit{efloorf}$$

$$\frac{t_1 \to t_1'}{\text{fl}(t_1) \to \text{fl}(t_1')}\ \mathit{edivd}$$

- Defina el sistema de tipos para esta extensión.

$$\frac{}{\Gamma \vdash \text{numr[r]: RealPositivo}}\ treal$$

$$\frac{\Gamma \vdash t_1\text{: RealPositivo} \quad \Gamma \vdash t_2\text{: RealPositivo}}{\Gamma \vdash div(t_1, t_2)\text{: RealPositivo}}\ tdiv$$

$$\frac{\Gamma \vdash t_1\text{: RealPositivo}}{\Gamma \vdash fl(t_1)\text{: Nat}}\ tfloor$$

Más las vistas en EAB

- Sea e = iszero(fl(2.5÷4.8)). Muestre paso a paso que
    - $\vdash$ e : Bool

$$\cfrac{\cfrac{\cfrac{\checkmark}{2{,}5\text{: PFloat}} \quad \cfrac{\checkmark}{4{,}8\text{: PFloat}}}{fl(2{,}5 \div 4{,}8)\text{: PFloat}}}{iszero(fl(2{,}5 \div 4{,}8))\text{: Bool}}$$

      $\therefore \vdash$ e : Bool

    - e $\to^*$ true
      e = iszero(fl(2.5÷4.8)).
      e $\to$ iszero(fl(numr[2.5/4.8])) $\to$ iszero(fl(0.5))
      $\to$ iszero(nat[floor(0.5)]) $\to$ iszero(nat[0])
      $\to$ true $\therefore$ e $\to^*$ true

[2]Los estados y los estados iniciales son los mismos de lo ya hecho en EAB.

[3]Sucesor y Predecesor son iguales a las ya definidas, al igual que las operaciones booleanas.

Ejercicio Semanal 6

Dos funciones son β-equivalentes syss para cada término r existe un término t tal que f$r \rightarrow_{\beta} * \ t \ \xi$ g$r \rightarrow_{\beta} * \ t$.
Demostrar que los pares de funciones son β-equivalentes.
Donde:
$\mathrm{I} = \lambda\mathrm{x.x}$
$\mathrm{Ap} = \lambda\mathrm{x}.\lambda\mathrm{y.xy}$
$\lambda\mathrm{x}.\lambda\mathrm{y}.<\mathrm{x, y}>= \lambda\mathrm{f}.\lambda\mathrm{s}.\lambda\mathrm{b.bfs}$
$\mathrm{w} = \lambda\mathrm{x.xx}$
$\mathrm{false} = \lambda\mathrm{x}.\lambda\mathrm{y.y}$

1. *I* y *Ap(Ap I)*. Sea r un término
   $I\mathrm{r} = \lambda\mathrm{x.x\ r} = \mathrm{r}$
   Por otro lado.
   $Ap(Ap\ I)\ r = \lambda\mathrm{x}.\lambda\mathrm{y.xy}(\lambda\mathrm{x}.\lambda\mathrm{y.xy}(\lambda\mathrm{x.x}))\mathrm{r}$
   $\lambda\mathrm{x}.\lambda\mathrm{y.xy}(\lambda\mathrm{x.x})\mathrm{r}$
   $\lambda\mathrm{x.x\ r}$
   r
   $\therefore I \ \xi \ Ap(Ap\ I)$ son β-equivalentes pues llegamos aplicando r al mismo término.

2. *Ap* y $\lambda\mathrm{x}.\lambda\mathrm{y}.<\mathrm{x, y}>\mathrm{I}$. Sea r un término
   $Ap\ r = \lambda\mathrm{f}.\lambda\mathrm{s.fs\ r} = \lambda\mathrm{s.rs}$
   Por otro lado.
   $\lambda x.\lambda y.<x,\ y>I = \lambda\mathrm{f}.\lambda\mathrm{s}.(\lambda\mathrm{b.bfs}\ (\lambda\mathrm{x.x}))\mathrm{r}$
   $(\lambda\mathrm{f}.\lambda\mathrm{s}(\lambda\mathrm{x.x})\mathrm{fs})\mathrm{r}$
   $(\lambda\mathrm{f}.\lambda\mathrm{s.fs})\mathrm{r}$
   $\lambda\mathrm{s.rs}$
   $\therefore Ap \ \xi \ \lambda x.\lambda y.<x,\ y>I$ son β-equivalentes pues llegamos aplicando r

al mismo término.

3. *I* y *w(w(false))* Sea r un término
Ir = λx.x r = r
Por otro lado.
λx.xx(λx.xx(λx.λy.y))r
λx.xx(λx.λy.y λx.λy.y)r
λx.xx(λy.y)r
(λy.y λy.y)r
λy.y r = r
$\therefore$ *I* ξ *w(w(false))* son β-equivalentes pues llegamos aplicando r al mismo término.

Ejercicio Semanal 7

Definir la función `length` para calcular la longitud de una lista recursivamente utilizando un combinador de punto fijo.
length = Y_g

g = λf.λl. if(isnil l) then 0 else 1 + f(tl l)
Donde:
'isnil' nos dice si una lista es vacía.
'tl' devuelve la cola de una lista.

Utilice su definición para calcular la longitud de la lista `(cons 3 (cons 2 nill))`.

length(cons 3 (cons 2 nil))
g((λx g(xx))(λx g(xx))) [3, 2]
$\longrightarrow$ if (isnil [3, 2]) then 0 else 1 + (λx g(xx))(λx g(xx)) (tl [3, 2])
1 + g(λx g(xx))(λx g(xx))
$\longrightarrow$ if(isnil [2]) then 0 else 1 + (λx g(xx))(λx g(xx)) (tl [2])
1 + g(λx g(xx))(λx g(xx))
$\longrightarrow$ if(isnil []) then 0 else 1 + (λx g(xx))(λx g(xx)) (tl [ ])
$\longrightarrow$ 1 + (1 + 0) = 2

Ejercicio Semanal 8

Usando el algoritmo W, verificar si las siguientes definiciones se pueden tipar.

- pair = λxyb.bxy = e
  Las llamadas son:
  W(e) $\leadsto$ W(λy.λb.bxy) $\leadsto$ W(λb.bxy) $\leadsto$ W(bxy) $\leadsto$ W(b), W(x), W(y).

W(y)	$\{y : \mathcal{Y}\} \vdash \{y : \mathcal{Y}\}$
W(x)	$\{x : \mathcal{X}\} \vdash \{x : \mathcal{X}\}$
W(b)	$\{b : \mathcal{B}\} \vdash \{b : \mathcal{B}\}$
W(bx)	$(\{b : \mathcal{B}\} \cup \{x : \mathcal{X}\})\mu \vdash (bx)\mu : V\mu, \mu = umg(\emptyset \cup \{\mathcal{B} = \mathcal{X} \to \mathcal{V}\})$ $(\{b : \mathcal{B}, x : \mathcal{X}\})\mu : V\mu, \mu = [\mathcal{B} := \mathcal{X} \to \mathcal{V}]$ $\{b : \mathcal{X} \to \mathcal{V}, x : \mathcal{X}\} \vdash bx : \mathcal{V}$
W(bxy)	$(\{b : \mathcal{X} \to \mathcal{V}, x : \mathcal{X}\} \cup \{y : \mathcal{Y}\})\mu \vdash (bxy)\mu : \mathcal{U}\mu, \mu = umg(\emptyset \cup \{\mathcal{V} = \mathcal{Y} \to \mathcal{U}\})$ $(\{b : \mathcal{X} \to \mathcal{V}, x : \mathcal{X}, y : \mathcal{Y}\})\mu \vdash (bxy)\mu : \mathcal{U}\mu, \mu = [\mathcal{V} := \mathcal{Y} \to \mathcal{U}]$ $\{b : \mathcal{X} \to (\mathcal{Y} \to \mathcal{U}), x : \mathcal{X}, y : \mathcal{Y} \vdash bxy : \mathcal{U}\}$
W(λb.bxy)	$\{b : \mathcal{X} \to (\mathcal{Y} \to \mathcal{U}), x : \mathcal{X}, y : \mathcal{Y}\} \setminus \{b : \mathcal{X} \to (\mathcal{Y} \to \mathcal{U})\} \vdash \lambda b : \mathcal{X} \to (\mathcal{Y} \to \mathcal{U}).bxy : (\mathcal{X} \to (\mathcal{Y} \to \mathcal{U})) \to \mathcal{U}$ $\{x : \mathcal{X}, y : \mathcal{Y}\} \vdash \lambda b : \mathcal{X} \to (\mathcal{Y} \to \mathcal{U}).\ bxy : (\mathcal{X} \to (\mathcal{Y} \to \mathcal{U})) \to \mathcal{U}$
W(λy.λb.bxy)	$\{x : \mathcal{X}, y : \mathcal{Y}\} \setminus \{y : \mathcal{Y}\} \vdash \lambda y : \mathcal{Y}.\lambda b : \mathcal{X} \to (\mathcal{Y} \to \mathcal{U}).\ bxy : \mathcal{Y} \to (\mathcal{X} \to (\mathcal{Y} \to \mathcal{U})) \to \mathcal{U}$
W(λx.λy.λb.bxy)	$\{x : \mathcal{X}\} \setminus \{x : \mathcal{X}\} \vdash \lambda x : \mathcal{X}.\lambda y : \mathcal{Y}.\lambda b : \mathcal{X} \to (\mathcal{Y} \to \mathcal{U}).bxy : \mathcal{X} \to \mathcal{Y} \to (\mathcal{X} \to (\mathcal{Y} \to \mathcal{U})) \to \mathcal{U}$

- fst = λp.p true = e
  Las llamadas son:

$W(e) \rightsquigarrow W(\lambda p.p \text{ true}) \rightsquigarrow W(p \text{ true}) \rightsquigarrow W(p), W(\text{true})$.
$W(\text{true}) \rightsquigarrow W(\lambda y.x) \rightsquigarrow W(x)$.

$W(p)$	$\{p : \mathcal{P}\} \vdash \{p : \mathcal{P}\}$
$W(\text{true})$	
$W(x)$	$\{x : \mathcal{X}\} \vdash \{x : \mathcal{X}\}$
$W(\lambda y.x)$	$\{x : \mathcal{X}\} \setminus \{y : \mathcal{Y}\} \vdash \lambda y : \mathcal{Y}.x : \mathcal{Y} \to \mathcal{X}$ $\{x : \mathcal{X}\} \vdash \lambda y : \mathcal{Y}.x : \mathcal{Y} \to \mathcal{X}$
$W(\lambda x.\lambda y.x)$	$\{x : \mathcal{X}\} \setminus \{x : \mathcal{X}\} \vdash \lambda x : \mathcal{X}.\lambda y : \mathcal{Y}.x : \mathcal{X} \to \mathcal{Y} \to \mathcal{X}$ $\vdash \lambda x : \mathcal{X}.\lambda y : \mathcal{Y}.x : \mathcal{X} \to \mathcal{Y} \to \mathcal{X}$
$W(p \text{ true})$	$(\{p : \mathcal{P}\} \cup \{\text{true} : \mathcal{X} \to \mathcal{Y} \to \mathcal{X}\})\mu : \mathcal{V}\mu$, $\mu = \text{umg}(\emptyset \cup \{\mathcal{P} = \mathcal{X} \to \mathcal{Y} \to \mathcal{X}\})$ $(\{p : \mathcal{P}, \text{true} : \mathcal{X} \to \mathcal{Y} \to \mathcal{X}\})\mu \vdash (p \text{ true})\mu : \mathcal{V}\mu$, $\mu = [\mathcal{P} := (\mathcal{X} \to \mathcal{Y} \to \mathcal{X}) \to \mathcal{V}]$ $\{p : (\mathcal{X} \to \mathcal{Y} \to \mathcal{X}) \to \mathcal{V}, \text{true} : \mathcal{X} \to \mathcal{Y} \to \mathcal{X}\} \vdash p \text{ true} : \mathcal{V}$
$W(\lambda p.p \text{ true})$	$\{p : (\mathcal{X} \to \mathcal{Y} \to \mathcal{X}) \to \mathcal{V}, \text{true} : \mathcal{X} \to \mathcal{Y} \to \mathcal{X}\} \setminus \{p : (\mathcal{X} \to \mathcal{Y} \to \mathcal{X}) \to \mathcal{V}\} \vdash \lambda p : (\mathcal{X} \to \mathcal{Y} \to \mathcal{X}) \to \mathcal{V}.p \text{ true} : (\mathcal{X} \to \mathcal{Y} \to \mathcal{X}) \to \mathcal{V} \to \mathcal{V}$ $\{\text{true} : \mathcal{X} \to \mathcal{Y} \to \mathcal{X}\} \vdash \lambda p : (\mathcal{X} \to \mathcal{Y} \to \mathcal{X}) \to \mathcal{V}.p \text{ true} : (\mathcal{X} \to \mathcal{Y} \to \mathcal{X}) \to \mathcal{V} \to \mathcal{V}$

- $\text{snd} = \lambda p.p \text{ false} = e$
  Las llamadas son:
  $W(e) \rightsquigarrow W(\lambda p.p \text{ false}) \rightsquigarrow W(p \text{ false}) \rightsquigarrow W(p), W(\text{false})$.
  $W(\text{false}) \rightsquigarrow W(\lambda y.y) \rightsquigarrow W(y)$.

W(p)	$\{p : \mathcal{P}\} \vdash \{p : \mathcal{P}\}$
W(false)	
W(y)	$\{y : \mathcal{Y}\} \vdash \{y : \mathcal{Y}\}$
W(λy.y)	$\{y : \mathcal{Y}\} \setminus \{y : \mathcal{Y}\} \vdash \lambda y : \mathcal{Y}.y : \mathcal{Y} \to \mathcal{Y}$ $\{x : \mathcal{X}\} \vdash \lambda y : \mathcal{Y}.x : \mathcal{Y} \to \mathcal{X}$
W(λx.λy.y)	$\{\} \setminus \{x : \mathcal{X}\} \vdash \lambda x : \mathcal{X}.\ \lambda y : \mathcal{Y}.y : \mathcal{X} \to \mathcal{Y} \to \mathcal{Y}$ $\vdash \lambda x : \mathcal{X}.\lambda y : \mathcal{Y}.y : \mathcal{X} \to \mathcal{Y} \to \mathcal{Y}$
W(p false)	$(\{p : \mathcal{P}\} \cup \{false : \mathcal{X} \to \mathcal{Y} \to \mathcal{Y}\})\mu : \mathcal{V}\mu,\ \mu = umg(\emptyset \cup \{\mathcal{P} = \mathcal{X} \to \mathcal{Y} \to \mathcal{Y}\})$ $(\{p : \mathcal{P}, false : \mathcal{X} \to \mathcal{Y} \to \mathcal{Y}\})\mu \vdash (p\ false)\mu : \mathcal{V}\mu,\ \mu = [\mathcal{P} := (\mathcal{X} \to \mathcal{Y} \to \mathcal{Y}) \to \mathcal{V}]$ $\{p : (\mathcal{X} \to \mathcal{Y} \to \mathcal{Y}) \to \mathcal{V}, false : \mathcal{X} \to \mathcal{Y} \to \mathcal{Y}\} \vdash p\ false : \mathcal{V}$
W(λp.p false)	$\{p : (\mathcal{X} \to \mathcal{Y} \to \mathcal{Y}) \to \mathcal{V}, false : \mathcal{X} \to \mathcal{Y} \to \mathcal{Y}\} \setminus \{p : (\mathcal{X} \to \mathcal{Y} \to \mathcal{Y}) \to \mathcal{V}\} \vdash \lambda p : (\mathcal{X} \to \mathcal{Y} \to \mathcal{Y}) \to \mathcal{V}.p\ false : (\mathcal{X} \to \mathcal{Y} \to \mathcal{Y}) \to \mathcal{V} \to \mathcal{V}$ $\{false : \mathcal{X} \to \mathcal{Y} \to \mathcal{Y}\} \vdash \lambda p : (\mathcal{X} \to \mathcal{Y} \to \mathcal{Y}) \to \mathcal{V}.p\ false : (\mathcal{X} \to \mathcal{Y} \to \mathcal{Y}) \to \mathcal{V} \to \mathcal{V}$

Ejercicio Semanal 9

Sean:

- btw = λx:Nat.λp:Nat$\times$(Nat$\times$Nat).((fst p <x) and x <(snd(snd(p))))
- e = btw 1 <pred(1), <0, suc(1)>>

Realizar:

- Expresar e utilizando sintáxis abstracta. Para expresar e, expresamos btw: btw = lam(Nat, x.(lam(prod(Nat, prod(Nat, Nat)), p.(and(lt(fst p, x))(lt(x, snd(snd(p))))))))
  e = app(app(lam(Nat, x.(lam(prod(Nat, prod(Nat, Nat)), p.(and(lt(fst p, x))(lt(x,snd(snd(p)))))))),num[1]), pair(num[1], pair(num[0],suc(num[1])))).

- Mostrar paso a paso que e $\rightarrow^*$ true.
  e = app(app(lam(Nat, x.(lam(prod(Nat, prod(Nat, Nat)), p.(and(lt(fst p, x))(lt(x,snd(snd(p)))))))),num[1]), pair(num[1], pair(num[0],suc(num[1])))).
  $\rightarrow$ app(lam p(and (lt (fst p, 1)), lt(1, snd(snd(p))))) pair(num[1], pair(num[0],suc(num[1]))).
  $\rightarrow$ and(lt (fst pair(num[1], pair(pred(num[1]), pair(num[0], suc(num[1])))),1), lt(num[1],snd(snd(pair(pred(num[1]), pair(num[0], suc(num[1])))))))).
  $\rightarrow$ and(lt(pred num[1], num[1]), lt(num[1], snd(snd(pair(pred(num[1]), pair(num[0], suc(num[1])))))))
  $\rightarrow$ and(lt(num[0], num[1]), lt(num[1], snd(snd(pair(pred(num[1]), pair(num[0], suc(num[1]))))))).
  $\rightarrow$ and(true, lt(num[1], snd(snd(pair(pred(num[1], pair(num[0], suc(num[1])))))))).
  $\rightarrow$ and(true, lt(num[1], snd(pair(num[0], suc(num[1]))))),
  $\rightarrow$ and(true, lt(num[1], suc(num[1]))),

$\rightarrow$ and(true, lt(num[1], num[2])).
$\rightarrow$ and(true, true).
$\rightarrow$ true.
$\therefore$ e $\rightarrow^*$ true.

- Mostrar paso a paso que $\{\} \vdash$ e : Bool

1. $\vdash$ x : Nat
2. $\vdash$ p : Nat $\times$ (Nat $\times$ Nat)
3. p : Nat $\times$ (Nat $\times$ Nat) $\vdash$ fst p : Nat (2)
4. p : Nat $\times$ (Nat $\times$ Nat) $\vdash$ snd p : Nat (2)
5. snd p : Nat $\vdash$ snd (snd (p)) : Nat (4)
6. fst p : Nat, x : Nat $\vdash$ fst p <x : Bool (1,3)
7. x : Nat, snd (snd (p)) : Nat $\vdash$ x <snd (snd (p)) : Bool (1,5)
8. fst p <x : Bool, x <snd (snd (p)) : Bool $\vdash$ and (fst p <x, x <snd (snd (p))) : Bool (6,7)
9. 1 : Nat, 0 : Nat
10. 1 : Nat $\vdash$ pred (1) : Nat (9)
11. 1 : Nat $\vdash$ succ (1) : Nat (9)
12. $\vdash$ pair (pred (1), pair (0, succ (1))) : Nat $\times$ (Nat $\times$ Nat) (9,10,11)
13. $\vdash$ λp.and((fst p <x), (x <(snd (snd (p))))) : Bool (8,3,12)
14. λx.λp.and((fst p <x), (x <(snd (snd (p))))) : Bool (1,3,9)

Ejercicio Semanal 10

Evaluar la llamada a la función div(4,2)

```
fun div (m : Nat, n: Nat): Nat =>
      let q = ref 0 in

            let r = ref m in
                  while !r >= n do
                        q := !q + 1;     = w_1
q_1 =                   r := !r - n;              = r_1
                  end_while
                  !q
            end

      end
```

$\longrightarrow <(), \text{fun div}(4,2) >$
$\longrightarrow <(), \text{let q} = \text{ref } 0 \text{ in } \mathrm{q}_1 >$
$\longrightarrow <(), \text{let q} = \ell_q \text{ in } \mathrm{r}_1 >$
$\longrightarrow <(\ell_q \mapsto 0), \text{let r} = \text{ref m in } \mathrm{w}_1\ [\mathrm{q} := \ell_q] >$
$\longrightarrow <(\ell_q \mapsto 0, \ell_r \mapsto 4), \text{let r} = \ell_r \text{ in } \mathrm{w}_1\ [\mathrm{q} := \ell_q] >$
$\longrightarrow <(\ell_q \mapsto 0, \ell_r \mapsto 4), \mathrm{w}_1\ [\mathrm{q} := \ell_q, \mathrm{r} := \ell_r] >$
$\longrightarrow <(\ell_q \mapsto 0, \ell_r \mapsto 4), \text{if !r} \geq 2 \text{ then } \ell_q := \text{!q} + 1;\ \ell_r := \text{!r - 2; } \mathrm{w}_1 \text{ else void}>$
$\longrightarrow <(\ell_q \mapsto 0, \ell_r \mapsto 4), \text{if } 4 \geq 2 \text{ then } \ell_q := \text{!q} + 1;\ \ell_r := \text{!r - 2; } \mathrm{w}_1 \text{ else void}>$
$\longrightarrow <(\ell_q \mapsto 0, \ell_r \mapsto 4), \ell_q := 0 + 1;\ \ell_r := 4 - 2;\ \mathrm{w}_1 >$
$\longrightarrow <(\ell_q \mapsto 0, \ell_r \mapsto 4), \ell_q := 1;\ \ell_r := 2;\ \mathrm{w}_1 >$
$\longrightarrow <(\ell_q \mapsto 1, \ell_r \mapsto 2), \mathrm{w}_1 >$
$\longrightarrow <(\ell_q \mapsto 1, \ell_r \mapsto 2), \text{if !r} \geq 2 \text{ then } \ell_q := \text{!q} + 1;\ \ell_r := \text{!r - 2; } \mathrm{w}_1 \text{ else void}>$
$\longrightarrow <(\ell_q \mapsto 1, \ell_r \mapsto 2), \text{if } 2 \geq 2 \text{ then } \ell_q := \text{!q} + 1;\ \ell_r := \text{!r - 2; } \mathrm{w}_1 \text{ else void}>$
$\longrightarrow <(\ell_q \mapsto 1, \ell_r \mapsto 2), \ell_q := 1 + 1;\ \ell_r := 2 - 2;\ \mathrm{w}_1 >$

$\longrightarrow$ <$(\ell_q \mapsto 1, \ell_r \mapsto 2)$, ℓ_q := 2; ℓ_r := 0; w_1 >
$\longrightarrow$ <$(\ell_q \mapsto 2, \ell_r \mapsto 0)$, w_1 >
$\longrightarrow$ <$(\ell_q \mapsto 2, \ell_r \mapsto 0)$, if !r $\geq$ 2 then ℓ_q := !q + 1; ℓ_r := !r - 2; w_1 else void>
$\longrightarrow$ <$(\ell_q \mapsto 2, \ell_r \mapsto 0)$, if 0 $\geq$ 2 then ℓ_q := !q + 1; ℓ_r := !r - 2; w_1 else void>
$\longrightarrow$ <$(\ell_q \mapsto 2, \ell_r \mapsto 0)$, void; !q >
$\longrightarrow$ <$(\ell_q \mapsto 2, \ell_r \mapsto 0)$, !q >
$\longrightarrow$ <$(\ell_q \mapsto 2, \ell_r \mapsto 0)$, 2 >

Ejercicio Semanal 11

ie Definimos:
restapos = λx:Nat λy:Nat.if(x < y) then raise(0) else (x - y)
e = handle(1 - (restapos 2 suc(2))) with x $\Rightarrow$ 2 * x
Evaluar e
$\Box \succ$ e
$\longrightarrow$ handle(–, x.prod(2,x)), $\Box \succ$ rest(1, restapos(2, suc(2)))
$\longrightarrow$ rest(–, restapos(2, suc(2))), handle(–, x.prod(2,x)), $\Box \succ$ 1
$\prec$ 1
$\longrightarrow$ rest(1, –), handle(–, x.prod(2,x)), $\Box \succ$ restapos(2, suc(2))
$\longrightarrow$ rest(1, –), handle(–, x.prod(2,x)), $\Box \succ$ if (lt(2, suc(2))), raise 0, rest(2, suc(2))
$\longrightarrow$ if(–, raise(0), rest(2, suc(2))), rest(1, –), handle(–, x.prod(2,x)), $\Box \succ$ lt(2, suc(2))
$\longrightarrow$ lt(–, suc(2)), if(–, raise(0), rest(2, suc(2))), rest(1, –), handle(–, x.prod(2,x)), $\Box \succ$ 2
$\prec$ 2
$\longrightarrow$ lt(2, –), if(–, raise(0), rest(2, suc(2))), rest(1, –), handle(–, x.prod(2,x)), $\Box \succ$ suc(2)
$\prec$ 3
$\prec$ true
$\longrightarrow$ rest(1, –), handle(–, x.prod(2,x)), $\Box$, *succ* raise(0)
$\longrightarrow$ raise(–), rest(1, –), handle(–, x.prod(2,x)), $\Box, \succ$ 0
$\prec$ 0
$\longrightarrow$ rest(1, –), handle(–, x.prod(2,x)), $\Box \ll$ raise(0)
$\longrightarrow$ handle(–, x.prod(2,x)), $\ll$ raise(0)
$\longrightarrow$ $\Box \succ$ prod(2,x) [x := 0]
$\longrightarrow$ prod(–, 0), $\Box \succ$ 2
$\prec$ 2

$$\begin{aligned}
&\longrightarrow \mathrm{prod}(2, -), \square \succ 0 \\
&\qquad\qquad\qquad\qquad\quad \prec 0 \\
&\longrightarrow \square \prec 0
\end{aligned}$$

Ejercicio Semanal 12

Evaluar (letcc k in 10 * (continue k 3)) + 2
$\square \succ$ suma(letcc k in 10 * (continue k 3), 2)
suma(–, 2), $\square \succ$ letcc k in 10 * (continue k 3)
$\underbrace{\text{suma(–, 2), } \square}_{\text{p}} \succ$ 10 * (continue k 3)[k := cont(p)]
prod(–, continue(cont(p), 3)), $\square \succ$ 10
$\prec$ 10
continue(–, 3), prod(10, –) $\square \succ$ cont(p)
$\prec$ cont(p)
continue(cont(p), –), prod(10, –) $\square \succ$ 3
$\prec$ 3
suma(3,–),$\square \succ$ 2
$\prec$ 2
$\square \prec$ 5

Transformar la siguiente función en Haskell a CPS, utilizar para la concatenación de [1,2], [5,4,3]

```
append [] ys = ys
append (x:xs) ys = x:(append xs ys)
```

cpsapp [ ] ys k = k ys
cpsapp (x:xs) ys k = cpsapp xs ys (λv.k(x:v))

cpsapp [1,2] [5,4,3] = cpsapp [2] [5,4,3] (λv.(λy.y)(1:v))
cpsapp [2] [5,4,3] cpsapp [ ] [5,4,3] (λv(λv(λy.y)(1:v)(2:v)))
(λv(λy.y)(1:2:v))

$$\begin{aligned}\text{cpsapp [ ] [5,4,3]} &= \lambda\text{v.}(\lambda\text{y.y})(1{:}2{:}\text{v})([5,4,3])\\ &\ (\lambda\text{y.y})(1{:}2{:}[5,4,3])\\ &\ 1{:}2{:}[5,4,3]\\ &\ [1,2,5,4,3]\end{aligned}$$

Ejercicio Semanal 13

Se tienen las siguientes relaciones:
R $\leq$ S, S $\leq$ T, T $\leq$ U para los siguientes tipos X, Y.
Determinar si se cumple X $\leq$ Y o Y $\leq$ X o ninguno.
X = R + B $\to$ $\{\ell_2$: U $\times$ T$\}$
Y = U + T $\to$ $\{\ell_1$: S + U, ℓ_2 : U $\times$ R, ℓ_3 : R $\times$ R$\}$

- X $\leq$ Y

$$\dfrac{\dfrac{\dfrac{\times}{U \leq R}}{U+T \leq R+B} \quad \{\ell_2 : U \times T\} \leq \{\ell_1 : S+U, \ell_2 : U \times R, \ell_3 : R \times R\}^{2}}{R+B \to \{\ell_2 : U \times T\} \leq U+T \to \{\ell_1 : S+U, \ell_2 : U \times R, \ell_3 : R \times R\}}$$

- Y $\leq$ X

$$\dfrac{\dfrac{\dfrac{\checkmark}{R \leq U} \quad B \leq T}{R+B \leq U+T} \quad \dfrac{\dfrac{\dfrac{\dfrac{\checkmark}{U \times U} \quad \dfrac{\times}{T \times R}}{U \times T \leq U \times R}}{\{\ell_2 : U \times T\} \leq \{\ell_2 : U \times R\}}}{\{\ell_2 : U \times T\} \leq \{\ell_1 : S+U, \ell_2 : U \times R, \ell_3 : R \times R\}}}{U+T \to \{\ell_1 : S+U, \ell_2 : U \times R, \ell_3 : R \times R \leq R+B \to \{\ell_2 : U \times T\}\}}$$

Ninguna se cumple.

[2] Como U $\leq$ R no se cumple, no es necesario continuar.

Parte II

Presenciales

Presencial 1

Dada la siguiente expresión, llenar la tabla con base en dicha expresión. El orden de las expresiones let, es, por supuesto, el orden de lectura como texto.

let x = let y = 2 * 2 in y end. in
    x + let z = 15 - x in 2 * z end.
end.

let	variable ligada	Exp. a ligar	alcance
1	x	let y = 2 * 2 in y end.	x + let 2 = 15 - x in 2 * 2 end.
2	y	2 * 2	y
3	z	15 - x	2 * z

Presencial 2

Recordemos:
cons = λhtb.bht
hd = λl.l true
tl = λl.l false
isnil = λl.l (λxy.false)
nil = λx.true

Evaluar hd(tl(tl(cons 3 (cons 2 (cons 1 nil)))))

Tomemos (cons 3 (cons 2 (cons 1 nil))) como l
hd(tl(λl.false))
hd(tl(cons 2(cons 1 nil)))
Tomemos (cons 2(cons 1 nil)) como l'
hd(λl'.l' false)
hd(cons 1 nil)
Tomemos (cons 1 nil) como l"
λl".l" true
1

Presencial 3

Recordemos:
$\widehat{0} = \lambda$x.x
$\widehat{n+1} =$ < false, $\widehat{n}$ >
Definir:

- S tal que S $\widehat{n} \longrightarrow \widehat{n+1}$
- P tal que P $\widehat{n+1} \longrightarrow \widehat{n}$

Evaluar S(S(S $\widehat{1}$))

S := λn.λb.b false n
P := λn.n false

S(S(S $\widehat{1}$))
S(S(< false, $\widehat{1}$ >))
S(< false, < false, $\widehat{1}$ >>)
< false, < false, < false, $\widehat{1}$ >>>

Presencial 4

Usando el algoritmo W, verificar si la siguiente expresión se puede tipar. e = (λx.λy.z)
W(λx.λy.z) $\rightsquigarrow$ W(λy.z) $\rightsquigarrow$ W(z)

W(z)	$\{z : \mathcal{Z}\} \vdash \{z : \mathcal{Z}\}$
W(λy.z)	$\{z : \mathcal{Z}\} \vdash \lambda y : \mathcal{Y}.z : \mathcal{Y} \to Z$
W(λx.λy.z)	$\{z : \mathcal{Z}\} \vdash \lambda\ x : \mathcal{X}.\lambda y : \mathcal{Y}.z : \mathcal{X} \to (\mathcal{Y} \to \mathcal{Z})$

Presencial 5

Con memoria inicial ($\ell_1 \mapsto 1$). Evaluar la llamada a la función e
e = let y = ref 2 in := !ℓ_1 - 1 end.
<($\ell_1 \mapsto 1$), e>
$\longrightarrow$ <($\ell_1 \mapsto 1$), let y = ref 2 in y := !ℓ_1 - 1 end.>
$\longrightarrow$ <($\ell_1 \mapsto 1$, $\ell_2 \mapsto 2$), let y = ℓ_2 in y := !ℓ_1 - 1 end.>
$\longrightarrow$ <($\ell_1 \mapsto 1$, $\ell_2 \mapsto 2$), y := !ℓ_1 - 1 [y := ℓ_2]>
$\longrightarrow$ <($\ell_1 \mapsto 1$, $\ell_2 \mapsto 2$), ℓ_2 := !ℓ_1 - 1>
$\longrightarrow$ <($\ell_1 \mapsto 1$, $\ell_2 \mapsto 2$), ℓ_2 := 1 - 1>
$\longrightarrow$ <($\ell_1 \mapsto 1$, $\ell_2 \mapsto 2$), ℓ_2 := 0>
$\longrightarrow$ <($\ell_1 \mapsto 1$, $\ell_2 \mapsto 0$), ()>

Presencial 6

Evaluar $\Box \succ$app ((λx.if (x + 1 > 0) then x * 2 else x),0)
app((–), 0), $\Box \succ$ λx.if (x + 1 > 0) then x * 2 else x)
$\prec$
app((λx.if (x + 1 > 0) then x * 2 else x), –), $\Box \succ$ 0
$\prec$ [x := 0]
$\Box \succ$ if (0 + 1) > 0 then x * 2 else x
if(–, x * 2, x), $\Box \succ$ 0 + 1 > 0
gt(–, 0), if(–, x * 2, x), $\Box \succ$ 0 + 1
sum(–, 1), gt(–, 0), if(–, x * 2, x), $\Box \succ$ 0
$\prec$ 0
sum(0, –), gt(–, 0), if(–, x * 2, x), $\Box \succ$ 0
$\prec$ 1
gt(1, –), if(–, x * 2, x), $\Box \succ$ 0
$\prec$ 0
if(–, x * 2, x), $\Box \prec$ true
$\Box \succ$ 0 * 2
mul(–, 2), $\Box \succ$ 0
$\prec$ 0
mul(0, –), $\Box \succ$ 2
$\prec$ 2
$\Box \prec$ 0

Presencial 7

Evaluar ((λw.letcc k in if w $>$ 0 then continue k 0 else 3) 4)

$\square \succ$ ((λw.letcc k in if w $>$ 0 then continue k 0 else 3) 4)
app(–, 4) , $\square \succ$ (λw.letcc k in if w $>$ 0 then continue k 0 else 3)
app((λw.letcc k in if w $>$ 0 then continue k 0 else 3), –), $\square \succ$ 4
$\prec$ [w := 4]
$\square \succ$ letcc k in if 4 $>$ 0 then continue k 0 else 3
letcc(k, –), $\square \succ$ if 4 $>$ 0 then continue k 0 else 3
if(–, continue k 0, 3), letcc(k, –), $\square \succ$ 4 $>$ 0
$\prec$ true
$\square \succ$ letcc(k. continue k 0)
$\square \succ$ continue k 0 [k := cont([])]
$\square \succ$ continue cont([]) 0
continue(–, 0), $\square \succ$ cont([])
$\prec$ cont([])
continue(cont([]), –), $\square \succ$ 0
$\prec$ 0
$\square \prec$ 0

Presencial 8

e = (λy. if (y == 0), then 1, else raise 0) 3

e' = handle e with x $\Rightarrow$ x - 1

Evaluar e'

$\square \succ$ e'

$\longrightarrow_{\mathcal{K}}$ handle(–, x . res(x - 1)); $\square \succ$ (λy. if (Eq y 0), then 1, else raise 0) 3

$\longrightarrow_{\mathcal{K}}$ $\prec$ y := 3

$\longrightarrow_{\mathcal{K}}$ if(–, then 1, else raise 0); handle(–, x. res(x - 1)); $\square \succ$ Eq 3 0

$\longrightarrow_{\mathcal{K}}$ $\prec$ false

$\longrightarrow_{\mathcal{K}}$ handle(–, x. res(x - 1)); $\square \succ$ raise 0

$\longrightarrow_{\mathcal{K}}$ $\ll$ x := 0

$\longrightarrow_{\mathcal{K}}$ res(–, -1); $\square \succ$ 0

$\longrightarrow_{\mathcal{K}}$ $\prec$ 0

$\longrightarrow_{\mathcal{K}}$ res(0, –); $\square \succ$ -1

$\longrightarrow_{\mathcal{K}}$ $\prec$ -1

$\longrightarrow_{\mathcal{K}}$ $\square \prec$ -1

Presencial 9

A $\leq$ B
Determinar si X $\leq$ Y o Y $\leq$ X o ninguno.
X = B $\rightarrow \{\ell : \text{A}\}$
Y = A $\rightarrow \{\ell : \text{A}\}$

- X $\leq$ Y

$$\dfrac{\dfrac{\checkmark}{A \leq B} \quad \dfrac{\dfrac{\checkmark}{A \leq A}}{\{\ell : A\} \leq \{\ell : A\}}}{B \rightarrow \{\ell : A\} \leq A \rightarrow \{\ell : A\}}$$

- Y $\leq$ X

$$\dfrac{\dfrac{\times}{B \leq A} \quad \dfrac{\dfrac{\checkmark}{A \leq A}}{\{\ell : A\} \leq \{\ell : A\}}}{A \rightarrow \{\ell : A\} \leq B \rightarrow \{\ell : A\}}$$

Se cumple X $\leq$ Y.

Parte III

Exámenes Parciales

Parcial 1

1. Considere el siguiente programa $\mathbb{P}$ en `POSTFIX`

   (postfix 3 mul swap 2 mul swap sub)

   a) Evaluar $\mathbb{P}$ siendo [2,3,5] la pila inicial.

$$[2, 3, 5]$$

$$[6, 5]$$

$$[5, 6]$$

$$[2, 5, 6]$$

$$[10, 6]$$

$$[6, 10]$$

$$[4]$$

b) Dada la pila arbitraria [x,y,z] ¿Qué calcula el programa $\mathbb{P}$?
Calcula el residuo de dos veces el tercer elemento (z) con el producto del primero (x) con el segundo (y).

2. En el lenguaje de programación `Zoo`, los nombres para las variables deben empezar con el caracter 'V' seguido por una cadena cualquiera no vacía de caracteres 'o' u 'z'.

a) Defina un juicio `ozv` tal que `s ozv` se cumple si y solo si `s` es un nombre válido de variable de `Zoo`.

$$\frac{}{\text{Vo ozv}}\ (1) \qquad \frac{}{\text{Vz ozv}}\ (2) \qquad \frac{\text{s ozv}}{\text{so ozv}}\ (3) \qquad \frac{\text{s ozv}}{\text{sz ozv}}\ (4)$$

b) Derive `Vozo ozv` usando las reglas.

$$\cfrac{\cfrac{\cfrac{}{\text{Vo ozv}}\ (1)}{\text{Voz ozv}}\ (4)}{\text{Vozo ozv}}\ (3)$$

c) Enuncie el principio de inducción estructural para el juicio `ozv` y utilícelo para demostrar que: si w *ozv* entonces $\exists$u$\in$\{o,z\}$^+$(w = `Vu`)
Base
Vo ozv
Vz ozv
H.I. Suponer que una cadena no vacía cumple `ozv`.
P.I. Demostrar que si w `ozv` ent. `wz ozv` o `wo ozv`
P.D. si $w \in$ `ozv`, ent. $\exists$v$\in$\{o,z\}$^+$(w = `Vu`)
base.

- sup w = Vo
  P.D. $\exists$v$\in$\{o,z\}$^+$(Vo = `Vu`) basta ver que u = o
- sup w = Vz
  P.D. $\exists$v$\in$\{o,z\}$^+$(Vz = `Vu`) basta ver que u = z

Hip. Sea $w \in$ `ozv`, ent. $\exists$u$\in$\{o,z\}$^+$(w = `Vu`)
P.I
P.D.

- Sup w = w'z
  por hip w' $\in$ `ozv`

por $\dfrac{\text{w' ozv}}{\text{w'z ozv}}$ (4), z $\in$ {o,z}$^+$
w = w'z `ozv`
$\rightarrow$ w `ozv`
$\rightarrow$ (w = Vu) (z = u)

- Sup w = w'o
por hip w' $\in$ `ozv`
por $\dfrac{\text{w' ozv}}{\text{w'o ozv}}$ (3), o $\in$ {o,z}$^+$
w = w'o `ozv`
$\rightarrow$ w `ozv`
$\rightarrow$ (w = Vu) (o = u)

3. Realice lo que se pide para la siguiente expresión e

```
let x = let y = 2 in y + 2 end in
   let x = x + 4 in
      let y = x * 2 in y + x end
   end
end
```

a) Llenar la siguiente tabla con base en e.

let	variable ligada	Exp. a ligar	alcance
1	x	let y = 2 in y + 2	let x = x + 4 in let y = x * 2 in y + x end end
2	y	2	y + 2
3	x	x + 4	let y = x * 2 in y + x end
4	y	x * 2	y + x

b) Encontrar la representación en la sintáxis abstracta de orden superior de e. Está permitido usar sintáxis concreta para operaciones y números.
let(let(2, y. y + 2), x. let(x + 4, x. let(x * 2, y. y + x)))

c) Encontrar una expresión e' que sea α-equivalente a e, y en donde todas las variables ligadas tengan distinto nombre.
let(let(2, a. a + 2), b. let(b + 4, c. let(c * 2, d. d + c)))

d) ¿Cuál es el valor de e? explicar como se llegó al resultado.
24. Asignamos a y 2, resolvemos y da 4, se lo asignamos a x, luego la segunda x es distinta, así que le asignamos la primera (4) más 4, tenemos 8; Luego asignamos la segunda y y tenemos 16, y al final solo sumamos 8 y llegamos a 24.

4. Chon Hacker desea extender al lenguaje de expresiones aritméticas EA con una instrucción para sumas asbitrarias, para o cual agrega la siguiente descripción al manual de usuario:

   - Sintáxis: *letsum* e_1 *with* **x** *from* e_2 *to* e_3 *end*
   - Semántica: Para evaluar una expresión *letsum* debemos evaluar e_2 y e_3 obtenienendo como resultados v_2 y v_3 respectivamente. El valor de la expresión *(*letsum) será entonces el valor de

     $$\sum_{x=v_2}^{v_3} e_1$$

     . Obsérvese que la variable `x` se considera ligada en la expresión e_1.
   - Ejemplos: La expresión e_1 = *letsum x with x from 1 * 2 to 2 + 2 end* se evalua a *2 + 3 + 4 = 9*; la expresión e_2 = 10 * (*letsum z * z with z from 3 + 1 to 8 - 2 end*) se evalúa a 10 * (16 + 25 + 36) = 770.

   a) Defina la sintáxis concreta mediante el juicio s E para expresión `letsum`.

   $$\frac{eE \quad vVar \quad aNat \quad bNat \quad a < b}{\text{letsum e with x from a to b}}$$

   b) Defina la sintáxis abstracta de orden superior para la expresión `letsum` mediante el juicio `ala`.

   $$\frac{t_1 ala \quad t_2 ala}{\text{letsum}(t_1.x, t_2) \text{ ala}}$$

   c) Escriba las expresiones *e1, e2* dadas en los ejemplos anteriores usando la sintáxis abstracta recién definida.
   letsum(x, x. prod(1,2) to sum(2,2))
   prod(10,letsum(prod(z,z), z. sum(3,1) to res(8, 2)))

5. Realizar la siguiente sustitución mostrando la respuesta paso a paso:
   (let y = x * 3 in (let z = y * 3 in x * y end) * 2 + x end) [x := 6 * y + z]
   (let a = x * 3 in (let b = a * 3 in x * a end) * 2 + x end) [x := 6 * y + z]
   (let a = (6 * y + z) * 3 in (let b = a * 3 in (6 * y + z) * a end) * 2 + (6 * y + z) end)

Parcial 2

1. Chon Hacker desea extender al lenguaje `EAB` con un operador ternario $<\bullet, \bullet, \bullet>$tal que <m, n, r >decide si los números *m, n*y *r* están en secuencia, es decir si $n = m + 1$ y $r = n + 1$. Por ejemplo <4, 10 - 5, 2 * 3 >devuelve ***true*** mientras que <2 + 3, 5 * 2, *suc 3* >devuelve ***false***.
   Defina lo siguiente para el operador $<\bullet, \bullet, \bullet>$.

   a) La sintáxis abstracta.
   $$\frac{t_1 \text{asa } t_2 \text{asa } t_3 \text{asa}}{sec(t_1, t_2, t_3)\text{asa}}$$

   b) La semántica estática.
   $$\frac{\Gamma \vdash n : \text{Nat } \Gamma \vdash m : \text{Nat } \Gamma \vdash r : \text{Nat}}{\Gamma \vdash sec(n, m, r) : \text{Bool}}$$

   c) La semántica dinámica, de las siguientes dos maneras:

      1) La estrategia de izquierda a derecha.

$$\frac{t_1 \to t_1'}{sec(t_1, t_2, t_3) \to sec(t_1', t_2, t_3)}$$

$$\frac{t_2 \to t_2'}{sec(num[n], t_2, t_3) \to sec(num[n], t_2', t_3)}$$

$$\frac{t_3 \to t_3'}{sec(num[n], num[m], t_3) \to sec(num[n], num[m], t_3')}$$

$$\frac{m = n + 1 \quad r = m + 1}{sec(num[n], num[m], num[r]) \to bool[true]}$$

$$\frac{}{sec(num[n], num[m], num[r]) \to bool[false]}$$

2) La estrategia que corta la ejecución sin evaular al tercer argumento en caso de que los primeros dos no cumplan estar en secuencia.

$$\frac{t_1 \to t_1'}{sec(t_1, t_2, t_3) \to sec(t_1', t_2, t_3)}$$

$$\frac{t_2 \to t_2'}{sec(t_1, t_2, t_3) \to sec(t_1, t_2', t_3)}$$

$$\frac{m = n + 1 \quad t_3 \to t_3'}{sec(num[n], num[m], t_3) \to sec(num[n], num[m], t_3')}$$

$$\frac{m = n + 1 \quad r = m + 1}{sec(num[n], num[m], num[r]) \to bool[true]}$$

$$\frac{}{sec(num[n], num[m], t_3) \to bool[false]}$$

d) Sea e = <*suc 2*, 1 + 6, 5 >. Verifique que e $\to^*$ false, siguiente la segunda estrategia. Puede obviar varios pasos pero no los que involucran al operador <•, •, •>.
e $\to$ <3, 1 + 6, 5 >
$\to$ <3, 7, 5 >
$\to$ false

e) Sea r = if *iszero* (y + x) then <y, *suc x*, 13 >else z. Verifique que se cumple: x :Nat, y :Nat, z :Bool $\vdash$ r :Bool

$$\dfrac{\dfrac{\dfrac{\Gamma\vdash y:\text{Nat}\quad \Gamma\vdash x:\text{Nat}}{\Gamma\vdash y+x:\text{Nat}}}{\Gamma\vdash iszero(y+x):\text{Bool}} \quad \dfrac{\Gamma\vdash y:\text{Nat}\quad \dfrac{\Gamma\vdash x:\text{Nat}}{\Gamma\vdash sucx:\text{Nat}}\quad \Gamma\vdash 13:\text{Nat}}{\Gamma\vdash <y, suc\ x, 13>:\text{Bool}} \quad \dfrac{}{\Gamma\vdash z:\text{Bool}}}{x:\text{Nat},\ y:\text{Nat},\ z:\text{Bool}\vdash r:\text{Bool}}$$

2. Los números de *Barendregt*, denotados como $\widehat{n}$, se definen como sigue:

$$\widehat{0} = \lambda x.x$$
$$\widehat{n+1} = \langle false, \widehat{n} \rangle$$

En esta definición recuerde que tanto *false* como la operación de par ordenado <>corresponden a cuertas abstracciones lambda defininas en clase.

Realice lo siguiente para los numerales de *Barendregt*:

a) Demuestre que $\widehat{n+1}0 \to_\beta^* \widehat{0}$ y que $\widehat{n+1}\widehat{m+1} \to_\beta^* \widehat{m}\widehat{n}$.

- $\widehat{n+1}\widehat{0} \to_\beta^* \widehat{0}$ (λb. false $\widehat{n}$)$\widehat{0}$
  $\to$ ($\widehat{0}$ false)$\widehat{n}$ $\to$ false $\widehat{n}$
  = $(\lambda xy.y)\widehat{n}\to$ $\lambda y.y = \widehat{0}$
- $\widehat{n+1}\widehat{m+1} \to_\beta^* \widehat{m}\widehat{n}$
  <false, $\widehat{n}$ ><false, $\widehat{m}$ >
  $\to$ (<false, $\widehat{m}$ false $\widehat{n}$)
  $\to$ ((false false $\widehat{m}$)$\widehat{n}$)
  $\to$ ($\widehat{m}$)$\widehat{n}$

b) Defina la función sucesor S y verifique que con su definición S$\widehat{n} \to^*$ $\widehat{n+1}$.
S := λn.<false, n >
= $\lambda n.\lambda b.b$ false n
S$\widehat{n}$ = (λn. <false, n >) $\widehat{n}$
$\to$ <false, $\widehat{n}$ >
= $\widehat{n+1}$

c) Defina la función predecesor P y verifique con su definición que P$\widehat{n+1} \to^* \widehat{n}$. ¿Cuál es la forma normal de P$\widehat{0}$?
P := λn. n false
P($\widehat{n+1}$) = (λn.n false) ($\widehat{n+1}$)

$\rightarrow (\widehat{n+1})$ false
$=$ <false, $\widehat{n}$ >false
$\rightarrow$ (false false. $\widehat{n}$)
$\rightarrow \widehat{n}$
P$\widehat{0}$ = (λn. false) $\widehat{0} \rightarrow \widehat{0}$ false $\rightarrow$ false

d) Defina la función test de cero Z y verifique que con su definición que Z$\widehat{n+1} \rightarrow^*$ *false* y que Z$\widehat{0} \rightarrow^*$ *true*.
Z := λn.n true
Z$(\widehat{n+1})$ = (λn. true) (false, $\widehat{n}$)
$\rightarrow$ (false, $\widehat{n}$) true
$\rightarrow$ (true false, $\widehat{n}$) $\rightarrow$ false
Z$\widehat{0}$ = (λn.n true) $\widehat{0}$
$\rightarrow \widehat{0}$ true $\rightarrow$ false

3. Sea e = z(($\lambda y.yx$) $\lambda z.yz$). Hallar un término e' α-equivalente a e de tal forma que las variables ligadas y libres sean distintas. Hallar la forma normal de e', mostrando la reducción paso a paso y subrayando el redex a evaluar.
e := z(($\lambda y.yx$) $\lambda z.yz$)
e := a$\underline{((\lambda b.bc)\ \lambda d.ed)}$
$\rightarrow$ a($\underline{(\lambda d.ed)\text{c}}$)
$\rightarrow$ a (ec)

4. Defina utilizando combinadores de punto fijo la función `reverse` que encuentre la reversa de una lista. Utilice su definición para mostrar que: `reverse (cons 3(cons 2 (cons 1 nil)))` $\rightarrow^*_\beta$ `cons 1( cons 2( cons 3 nil))` .
Puede suponer definidas y correctas todas las funciones de listas que requiera, salvo `reverse` por supuesto.
reverse := Yg
g = $\lambda f.\lambda l$ if (isnil l) then l else (append (hd l) (f(tl l)))
Y := $\lambda g.(\lambda x.$g (xx))($\lambda x.$g (xx))
$\rightarrow$ g(Yg)
reverse [3,2,1] = (g (Yg)) [3,2,1]
if (isnil [3,2,1]) then [3,2,1] else (append 3 (g (Yg)) [2,1])
reverse [2,1]
if (isnil [2,1]) then [2,1] else (append 2 (g (Yg)) [1])
reverse[1]
if (isnil [1]) then [1] else (append 1 (g (Yg)) [])
reverse[]

if (isnil []) then []
→ [1,2,3]

5. Sea G = $\lambda x.\lambda y.y(xy)$. Demuestre que si F $\rightarrow_\beta^*$ GF entonces F es un combinador de punto fijo.
   G := $\lambda x.\lambda y.y(xy)$
   Queremos ver que F $\rightarrow_\beta^*$ GF
   Fg $\rightarrow_\beta^*$ (GF)g
   = ($\lambda x.\lambda y.y(xy)$F)g
   → ($\lambda y.y(Fg)$)g
   → g(FG)

6. Realice lo que se pide para la siguiente expresión e:

```
10 * let x = let v = 4 in v * 2 end in
   let z = x - 1 in
     3 + let x = z + x in 3 * x end
   end
end
```

   a) Llenar la siguiente tabla con base en e.

let	variable ligada	Exp. a ligar	alcance
1	x	let v = 4 in v * 2 end	let z = x + 1 in 3 +
			let x = z + x in 3 * x end end
2	v	4	v * z
3	z	x - 1	3 + let x = z + x in 3 * x end
4	x	z + x	3 * x

   b) Encuentre la representación en la sintáxis abstracta de orden superior de e. Está permitido usar sintáxis concreta para operaciones y números.
      let(x. let(v.4, v * z), let(z. x - 1, 3 + let(x. z + x, 3 * x)))

   c) Encuentre una expresión e' que sea α-equivalente a e y en donde todas las variables ligadas tengan distinto nombre.
      let(a. let(b.4, b * z), let(c. a - 1, 3 + let(d. c + a, 3 * d)))

Parcial 3

1. Sea e = λ z. λ y. (λ x. xy)zy. Encuentre, mediante el algoritmo W, un tipo T y un término e^a tales que:

$$\vdash e^a : T \text{ y } erase(e^a) = e$$

W(x)	x: $\mathcal{X} \vDash$ x : X
W(y)	y: $\mathcal{Y} \vDash$ y : Y
W(z)	z: $\mathcal{Z} \vDash$ z : Z
W(xy)	$\{$x : $\mathcal{X}, y : \mathcal{Y}\} \vdash$ xy : $\mathcal{V}$ $\{$x : $\mathcal{Y} \to \mathcal{V}$, y : $\mathcal{Y}\} \vdash$ xy : $\mathcal{V}$
W(λx.xy)	$\{$y : $\mathcal{Y}\} \vdash$ (λx : $\mathcal{Y} \to \mathcal{V}$).xy : $(\mathcal{Y} \to \mathcal{V}) \to \mathcal{V}$
W((λx.xy)z)	z : $(\mathcal{Y} \to \mathcal{T})$ ((λx : $\mathcal{Y} \to \mathcal{V}$).xy : $(\mathcal{Y} \to \mathcal{V}) \to \mathcal{V}$) z : $\to \mathcal{Z}$
W((λx.xy)z)y	((λx : $\mathcal{Y} \to \mathcal{V}$).xy : $(\mathcal{Y} \to \mathcal{V}) \to \mathcal{V}$) xy : $\mathcal{T}$
W(λy.(λx.xy)zy)	(λy : $\mathcal{Y}$.(λx : $\mathcal{Y} \to \mathcal{V}$.xy)zy) : $\mathcal{Y} \to \mathcal{T}$
W(λx...)	λx : $\mathcal{Y} \to (\mathcal{X} \to \mathcal{T})$. ($\lambda$y : $\mathcal{Y}$.(λx : $\mathcal{Y} \to \mathcal{V}$.xy)zy) : $\mathcal{Y} \to \mathcal{T} \to \mathcal{Y} \to \mathcal{T}$

2. Suponga definido el tipo `Nat`. Considere el siguiente pseudocódigo estilo `HASKELL`

```
data HNum = Cero | Ent Nat | Rat (Nat, Nat)

conv :: HNum → HNum
conv Cero = Cero
conv (Ent n) = Rat (n, 1)
conv (Rat (n, m)) = Ent (2^n * 3^m)
```

a) Defina al tipo `HNum` y sus constructores usando tipo suma. *La suma de tipos asocia a la izquierda.*
(Void + Nat) + (Nat + Nat)
fun Cero() $\Rightarrow$ $\text{inl}_{\text{Nat + Nat}}$ (inl_{Nat}(l))
fun Ent(n : Nat) $\Rightarrow$ $\text{inl}_{\text{Nat + Nat}}$ (inr_{void}(n))
fun Rat (n m : Nat) $\Rightarrow$ $\text{inr}_{\text{void + Nat}}$ ((n,m))

b) Defina al tipo `HNum` y sus constructores usando tipos variante.
HNum := [Cero :Void, Ent : Nat, Rat : Nat $\times$ Nat]

c) Usando la definición del inciso *a)*, implemente la función `conv` mediante el operador de análisis de casos `case`.
conv Cero = Cero
conv (Ent n) = Rat(n,1)
conv (Rat(n,m)) = Ent(2^n * 3^m)

fun conv (h :HNum) $\Rightarrow$ case h of {
  $\text{inl}_{NatxNat}$(p) $\Rightarrow$ case p of {
    inl_{Nat}() $\Rightarrow$ $\text{inl}_{Nat+Nat}$(inl_{Nat}())
    inr_{void}() $\Rightarrow$ $\text{inl}_{void+Nat}$((n, 1))
  }
  $\text{inr}_{voidxNat}$(p) $\Rightarrow$ $\text{inl}_{Nat+Nat}$(inr_{void}(2^{fstp} * 3^{snp}))
}

3. Definauna función `swap : Ref T` $\rightarrow$ `Ref T` $\rightarrow$ `Void` tal que `swap x y` cause el efecto de intercambiar los valores guardados en las celdas `x, y`. Es decir.

$$\langle (l_x \mapsto v_1, l_2 \mapsto v_2), swapl_x y\ \rangle \rightarrow^* \langle (l_x \mapsto v_2, l_y \mapsto v_1), ()\ \rangle$$

Además realice lo siguiente.

a) Verifique el tipado de *swap* $l_x l_y$. Para esto proponga contextos adecuados Σ y Γ.
let t = !x in
  x := !y;
  y := t
end
Σ = {l_x : T, t_y : T}
Γ = {t : T}

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : RefT} \quad \dfrac{y : RefT}{!y : T}}{x := y : Void} \quad \dfrac{\dfrac{}{y : RefT} \quad \dfrac{}{t : T}}{y := t : Void}}{x = y, y = t : Void}}{let...end : Void}}{\Gamma|\Sigma \vdash (\lambda x RefT.\lambda y : RefT.let...end) : RefT \to RefT \to Void}$$

b) Defina una versión de `swap`, pero suponiendo que existe una instrucción de asignación de pares $\langle e_1, e_2 \rangle := \langle e_1', e_2' \rangle$ cuyo efecto consiste en ejecutar simultáneamente las asignaciones $e_1 := e_1', e_2 := e_2'$ ¿Qué ventaja tiene esta nueva versión de `swap` sobre la anterior?
fun swap(x y : RefT) $\Rightarrow$
sp = λm.λn <m, n >:= <!n, !m >
Ventaja: No hay que usar memoria extra para poder realizar el swap.

4. Considere el siguiente programa P:

```
let d = λx. x := 2*x in
let m = l_n - 1 in
  while m > 0 do
    d l_n;
    m := m - 1
  end_while
  l_n := m;
end
end
```

a) Escriba a P usando refencias explícitas. Nota importante: obseve que en las líneas 4,5 de P NO se están modificando a la misma celda de memoria.

```
let d = ref λx. !x := 2*!x in
let m = ref l_n - 1 in
  while !m > 0 do
    d l_n;
    !m := !m - 1
  end_while
  l_n := !m;
end
end
```

b) Evaluar P formalmente, siendo $\mu = (l_n \mapsto 2)$ la memoria inicial. Puede omitir algunos pasos pero debe dar todos aquellos que involucran cambios en la memoria y los relacionados a la instrucción while. En este último caso no es necesario usar la definición de la semántica operacional del while, basta con utilizar sus propiedades.
<($l_n \mapsto 2$), let d = ref λx !x := 2 * !x in e_1>
<μ, e_1[d := λx !x := 2 * !x] >
<μ, let m = ref l_n - 1 in e_2 >
<($l_1 \mapsto 2$, $l_m \mapsto 1$), e_2 [m := l_m] >
<μ', while 1 > 0 do λx. !x := 2 * !x l_n; !m = 0 end_while >
con w = λx. !x := 2 * !x l_n; !m = 0
<μ', if 1 > 0 then w else () >
<μ' l_n := 2 * !l_n; l_m = 0 while >
<μ' l_n = 2 * 2; l_m = 0; while >
<($l_n \mapsto 4$, $l_m \mapsto 0$), if !l_m > 0 then e_2 w(e_1 e_2) else ()>
<μ' if 0 > 0 then $e_2$while (e_1 e_2) else () >
<μ', (); l_n := m >
<($l_n \mapsto 4$, $l_m \mapsto 0$), l_n := !l_m >
<($l_n \mapsto 4$, $l_m \mapsto 0$), l_n := 0 >
<($l_n \mapsto 4$, $l_m \mapsto 0$) >

5. Chu Ba Ka Hacker agrega al manual de usuario de un lenguaje imperativo la siguiente instrucción de declaración de variables

- Sintáxis: *newvar x := e1 in e2*
- Semántica: *newvar x := e1 in e2* inicializa la variable x con el valor de *e1*, acto seguido ejecuta el comando *e2* y restaura el valor original de x. Por ejemplo si en la memoria inicial x valía 2 y y valía 5. entonces al terminar la ejecución de `newvar x := !x + 1 in y := !x*3 end` la memoria indica que x vale 2 y y vale 9.

a) Extienda la semántica operacional con esta nueva instrucción.
$$\frac{e_1 \to e_2}{< \mu, e_1 ine_2 \to < \mu, e_1' ine_2}$$

b) Extienda la semántica estática con esta nueva instrucción.
$$\frac{e_1 : Void \quad e_2 : Void}{\Gamma \vdash \text{newvar}x := e_1 ine_2 : Void}$$

6. Sea e:

fun *mist* (x : Nat, y : Nat) : Nat $\Rightarrow$ if $x < y$ then y else *mist* x *(y * 2)* end.

a) Reescriba a e en sintáxis abstracta con un operador fix.
fix(Nat, f. $\lambda x.\lambda y$. if lt(x, y) then y else f x prod(y 2))

b) Muestre que $\vdash e$: Nat.

$$\dfrac{\dfrac{\dfrac{x : \text{Nat} \quad y : \text{Nat}}{x < y : \text{Bool}} \quad y : \text{Nat} \quad \dfrac{z : \text{Nat} \quad y : \text{Nat}}{z * y : \text{Nat}}}{\text{if lt(x, y) then y else f x prod(y 2) : Nat}}}{\vdash e : \text{Nat}}$$

c) Muestre que app(app(e, 3), 2) $\rightarrow^*$ 4. Puede saltarse pasos menos los que involucren la evaluación del fix.
app(app(fix(Nat, f $\lambda x.\lambda y$. if x < y then y else f x (2 * y))3)2)
app(fix(Nat, f λy. if 3 < y then y else f 3 (2 * y))2)
fix(Nat, f if 3 < 2 then 2 else f 3 (2 * 2))
if (3 < 2 then 2 else fix (3 4))
app(app(e, 3,4))
fix(Nat, f $\lambda x.\lambda y$. if x < y then y else f x (2 * y))3, 4
if (3 < 4 then 4 else fix 3 (2 * 4))
$\therefore$ app(app(e, 3)2) $\rightarrow^*$ 4

Parte IV

Laboratorio

Todos los programas hechos en el laboratorio son interpretados por `GHCI`, la idea es que se pueden ejecutar de la siguiente forma. Digamos que se quiere ejecutar el archivo de nombre `Practica1.hs`.

Lo que hacemos es en la carpeta en donde se tenga el archivo ejecutar:

```
\$:ghci Practica1.hs
```

Ejercicio Semanal 1

En este semanal vemos dos tipos de datos, las listas Snoc, y números naturales con su representación en binario.

0.1. Listas Snoc

```
data ListS a = NilS | Snoc (ListS a) a deriving Show
```

Un ejemplo de una lista `Snoc` es la lista [1,2,3,4,5] se ve la siguiente manera.

```
Snoc(Snoc(Snoc(Snoc(Snoc NilS 1)2)3)4)5
```

Las siguientes son funciones recursivas para el tipo de dato ListS(`Listas Snoc`).

```
-- | headS. Funcion que obtiene el primer elemento de la
↪  lista.
headS :: ListS a -> a
headS l = case l of
  NilS -> error "Empty list"
  (Snoc l' x) -> case l' of
    NilS -> x
    (Snoc l'' y) -> headS l'

-- | tailS. Funcion que obtiene la lista sin el primer
↪  elemento.
```

```haskell
tailS :: ListS a -> ListS a
tailS l = case l of
  NilS -> error "Empty list"
  (Snoc l' x) -> case l' of
    NilS -> NilS
    (Snoc l'' y) -> Snoc (tailS l') x

-- | initS. Funcion que obtiene la lista sin el primer
↪  elemento.
initS :: ListS a -> ListS a
initS l = case l of
  NilS -> error "Empty list"
  (Snoc l' _) -> l'

-- | lastS. Funcion que obtiene el ultimo elemento de la
↪  lista.
lastS :: ListS a -> a
lastS l = case l of
  NilS -> error "Empty list"
  (Snoc _ x) -> x

-- | nthElementS. Funcion que regresa el n-esimo elemento de
↪  la lista.
nthElementS :: Int -> ListS a -> a
nthElementS n l = case n of
  0 -> headS l
  n -> if n < 0
       then error "Invalid index"
       else case l of
              NilS -> error "Invalid index"
              l -> nthElementS (n-1) (tailS l)

-- | deleteNthElementS. Funcion que elimina el n-esimo
↪  elemento de la lista.
deleteNthElementS :: Int -> ListS a -> ListS a
deleteNthElementS n l = if n > longS l
                        then NilS
                        else if n < 0
                             then error "Invalid index"
                             else deleteNthElementSAux n l
```

```haskell
-- | addFirstS. Funcion que obtiene la lista donde el
-- |            primer elemento es el elemento dado.
addFirstS :: a -> ListS a -> ListS a
addFirstS x l = case l of
  NilS -> Snoc NilS x
  (Snoc l' y) -> Snoc (addFirstS x l') y

-- | addLastS. Funcion que obtiene la lista donde el
-- |           ultimo elemento es el elemento dado.
addLastS :: a -> ListS a -> ListS a
addLastS x l = case l of
  NilS -> Snoc NilS x
  (Snoc l' y) -> Snoc (Snoc l' y) x

-- | reverseS. Funcion que obtiene la reversa de la
-- |           lista.
reverseS :: ListS a -> ListS a
reverseS l = case l of
  NilS -> NilS
  (Snoc l x) -> Snoc (reverseS (tailS (Snoc l x)))
                     (headS  (Snoc l x))

-- | appendS. Funcion que obtiene la concatenación de dos
-- |          listas.
appendS :: ListS a -> ListS a -> ListS a
appendS l l2 = case l of
  NilS -> l2
  (Snoc l' x) -> case l2 of
    NilS -> l
    (Snoc l'' y) -> appendS (addLastS(headS(Snoc l'' y))
                            (Snoc l' x)) (tailS(Snoc l'' y))

-- | takeS. Funcion que obtiene la lista con los primeros
-- |        n elementos.
takeS :: Int -> ListS a -> ListS a
takeS n l = if longS l < n
            then l
            else case n of
                   0 -> NilS
```

```
                    n -> case l of
                      NilS -> NilS
                      (Snoc l x) -> addFirstS (headS l)
                           (takeS (n-1) (tailS (Snoc l x)))
```

0.2. Números naturales

```
data Nat = Zero | D (Nat) | O (Nat) deriving Show
```

Recordemos que Zero es la represantacion del cero (0), D x representa al doble de x, con x un número natural (2x), y O x representa al sucesor del doble de x, con x un número natural (2x + 1). Un ejemplo de la represantacion de un número natural en binario es 10011 que lo vemos de esta forma:

```
(O ( O ( D ( D ( O Zero)))))
```

Las siguientes son funciones recursivas para el tipo de dato `Nat`(`Numeros naturales`).

```
-- | toNat. Funcion que obtiene la representacion en numeros
-- |        Nat de un numero entero.
toNat :: Int -> Nat
toNat 0 = Zero
toNat n = if n `mod` 2 == 0
          then D(toNat (div n 2))
          else O(toNat (div (n-1) 2))

-- | succ. Funcion que obtiene el sucesor de un numero Nat.
succN :: Nat -> Nat
succN n = case n of
  Zero -> O Zero
  (O x) -> D(succN x)
  (D x) -> (O x)

-- | pred. Funcion que obtiene el predecesor de un numero
↪  Nat.
```

```
predN :: Nat -> Nat
predN n = case n of
  Zero -> Zero
  O Zero -> Zero
  (O x) -> D x
  (D x) -> O(predN x)

-- | add. Funcion que obtiene la suma de dos numeros Nat.
addN :: Nat -> Nat -> Nat
addN :: Nat -> Nat -> Nat
addN n m = case n of
  Zero -> m
  n -> case m of
    Zero -> n
    m -> addN (succN n) (predN m)

-- | prod. Funcion que obtiene el producto de dos numeros
↪  Nat.
prod :: Nat -> Nat -> Nat
prod Zero _ = Zero
prod _ Zero = Zero
prod n m =
  let
    n1 = mataD n
    m1 = mataD m
  in
    addN m1 (prod (predN n1) m1)
```

Las siguientes son funciones auxiliares.

```
deleteNthElementSAux :: Int -> ListS a -> ListS a
deleteNthElementSAux n l = case n of
  0 -> tailS l
  n -> case l of
    NilS -> NilS
    (Snoc l x) -> addFirstS (headS l) (deleteNthElementSAux
    ↪  (n-1) (tailS (Snoc l x)))

longS :: ListS a -> Int
```

```
longS NilS = 0
longS (Snoc l x) = 1 + longS l

mataD :: Nat -> Nat
mataD d = case d of
  Zero -> Zero
  D Zero -> Zero
  O Zero -> O Zero
  (D x) -> D(mataD x)
  (O x) -> O(mataD x)
```

Práctica 1

0.3. Postfix

En esta práctica vemos una implementación de *Postfix*, que es una secuencia parentizada que consiste en una palabra reservada `Postfix`, seguida de un número natural que indica el número de argumentos que recibe el programa, seguido de cero o más comandos.

```
data PF = POSTFIX deriving (Show, Eq)

data Command = I Int | ADD | DIV | Eq | EXEC | Gt | Lt
             | MUL | NGET | POP | REM | SEL | SUB | SWAP
             | ES [Command] deriving(Show, Eq)
```

Con los siguientes tipos.

```
type Program = (PF, Int, [Command])

type Stack = [Command]
```

Un ejemplo de un programa en Postfix:

```
(POSTFIX, 0, [ES[I 0, SWAP, SUB], I 7, SWAP, EXEC]) []
```

Las siguientes son funciones para la implementación de Postfix, algunas son recursivas.

```haskell
-- | arithOperation. Funcion que realiza las operaciones de
-- |                 los comandos aritmeticos. (add, div,
-- |                 eq, gt, lt, mul, rem, sub).
arithOperation :: Command -> Command -> Command -> Command
arithOperation (I n) (I m) com = case com of
                                   ADD -> I (m + n)
                                   DIV -> if m == 0
                                          then error
                                           ↪  "Division
                                           ↪  entre 0"
                                          else I (div n m)
                                   Eq  -> if n == m
                                          then I 1
                                          else I 0
                                   Gt  -> if n < m
                                          then I 1
                                          else I 0
                                   Lt  -> if m < n
                                          then I 1
                                          else I 0
                                   MUL -> I (m * n)
                                   REM -> I (mod n m)
                                   SUB -> I (n - m)
arithOperation _ _ _ = error "Error en tipo de datos."

-- | stackOperation. Funcion que realiza las operaciones de
-- |                 los comandos que alteran la pila de
-- |                 valores.(ADD, DIV, Eq, Gt, Lt, MUL,
-- |                 REM, SUB).
stackOperation :: Stack -> Command -> Stack
stackOperation s com = case com of
                         (I n) -> [(I n)] ++ s
                         ES xs -> [ES xs] ++ s
                         POP   -> if (validaStack s com)
                                  then cola s
                                  else error "Error en
                                   ↪  Stack, faltan
                                   ↪  elementos"
                         SWAP  -> if (validaStack s com)
                                  then swapiaux s
```

```
                                      else error "Error en
                                       ↪  Stack, faltan
                                       ↪  elementos"
                             SEL   -> if (validaStack s com)
                                      then selaux s
                                      else error "Error en
                                       ↪  Stack, faltan
                                       ↪  elementos"
                             NGET  -> if(validaStack s com)
                                      then ngetaux s
                                      else error "Error en
                                       ↪  Stack, faltan
                                       ↪  elementos"

-- | execOperation. Funcion que devuelve la lista de
-- |                comandos y la pila resultante de
-- |                realizar la llamada a la operacion
-- |                con exec.
execOperation :: [Command] -> Stack -> ([Command], Stack)
execOperation c ((ES x):y) = (x++c,y)
execOperation _ _ = error "UPS"

-- | validProgram. Funcion que determina si la pila de
-- |               valores que se desea ejecutar con un
-- |               programa es valido.
validProgram :: Program -> Stack -> Bool
validProgram (pf,n,l) s = (valid pf n l) && (n == length s)
 ↪  && (litS s)

-- | executeCommands. Funcion que dada una lista de comandos
-- |                  y una pila de valores obtiene la pila
-- |                  de valores resultantes después
 ↪  ejecutar
-- |                  todos los comandos.
executeCommands :: [Command] -> Stack -> Stack
executeCommands [] s = s
executeCommands (c:cs) s = case c of
                             (I x) -> executeCommands cs ((I
                              ↪  x):s)
                             ADD -> if (length s) >= 2
```

```
  then executeCommands cs
   ↪  ((arithOperation ((fst
   ↪  (splitAt 2 s))!!1) ((fst
   ↪  (splitAt 2 s))!!0)
   ↪  c):(snd (splitAt 2 s)))
  else error "Not enough
   ↪  numbers to add"
MUL -> if (length s) >= 2
  then executeCommands cs
   ↪  ((arithOperation (fst
   ↪  (splitAt 2 s)!!1) (fst
   ↪  (splitAt 2 s)!!0) c):snd
   ↪  (splitAt 2 s))
  else error "Not enough
   ↪  numbers to multiply"
DIV -> if (length s) >= 2
  then executeCommands cs
   ↪  ((arithOperation (fst
   ↪  (splitAt 2 s)!!1) (fst
   ↪  (splitAt 2 s)!!0) c):snd
   ↪  (splitAt 2 s))
  else error "Not enough
   ↪  numbers to divide"
SUB -> if (length s) >= 2
  then executeCommands cs
   ↪  ((arithOperation (fst
   ↪  (splitAt 2 s)!!1) (fst
   ↪  (splitAt 2 s)!!0) c):snd
   ↪  (splitAt 2 s))
  else error "Not enough
   ↪  numbers to subtract"
REM -> if (length s) >= 2
       then executeCommands cs
        ↪  ((arithOperation
        ↪  (fst (splitAt 2
        ↪  s)!!1) (fst (splitAt
        ↪  2 s)!!0) c):snd
        ↪  (splitAt 2 s))
       else error "Not enough
        ↪  numbers to ..."
```

```
Gt -> if (length s) >= 2
      then executeCommands cs
       ↪  ((arithOperation (fst
       ↪  (splitAt 2 s)!!1)
       ↪  (fst (splitAt 2
       ↪  s)!!0) c):snd
       ↪  (splitAt 2 s))
      else error "Not enough
       ↪  numbers to compare"
Lt -> if (length s) >= 2
      then executeCommands cs
       ↪  ((arithOperation (fst
       ↪  (splitAt 2 s)!!1)
       ↪  (fst (splitAt 2
       ↪  s)!!0) c):snd
       ↪  (splitAt 2 s))
      else error "Not enough
       ↪  numbers to compare"
Eq -> if (length s) >= 2
      then executeCommands cs
       ↪  ((arithOperation (fst
       ↪  (splitAt 2 s)!!1)
       ↪  (fst (splitAt 2
       ↪  s)!!0) c):snd
       ↪  (splitAt 2 s))
      else error "Not enough
       ↪  numbers to compare"
SWAP -> if (length s) >= 2
        then executeCommands cs
         ↪  ((stackOperation s
         ↪  c))
        else error "Not enough
         ↪  arguments to swap."
(ES xs) -> executeCommands cs
 ↪  (stackOperation s (ES xs))
EXEC ->  executeCommands
 ↪  (execaux s ++ cs) (cola s)
POP -> if (length s) >= 2
```

```
                                    then executeCommands cs
                                    ↪  ((stackOperation s
                                    ↪  c))
                                    else error "Not enough
                                    ↪  arguments."
                             SEL -> if (length s) >= 2
                                    then executeCommands cs
                                    ↪  ((stackOperation s
                                    ↪  c))
                                    else error "Not enough
                                    ↪  arguments."
                             NGET -> if (length s) >= 2
                                     then executeCommands cs
                                     ↪  ((stackOperation s
                                     ↪  c))
                                     else error "Not enough
                                     ↪  arguments."

-- | executeProgram. Función que ejecuta cualquier programa
-- |                 en Postfix.
executeProgram :: Program -> Stack -> [Command]
executeProgram (pf, i, k) s = if validProgram (pf,i,k) s
                              then executeCommands k s
                              else error "NO es un programa
                              ↪  valido."
```

Las siguientes son funciones auxiliares que se usaron en la resolución de la práctica.

```
selaux :: Stack -> Stack
selaux (x:y:z:zs) = if (comToNat z == 0)
                    then (x:zs)
                    else (y:zs)

comToNat :: Command -> Int
comToNat (I n) = n
comToNat k = error "Tipo de dato no aceptado en SEL"

isNat :: Command -> Bool
```

```
isNat (I n) = True
isNat k = False


valid :: PF -> Int -> [Command] -> Bool
valid pf i xs = if (pf == POSTFIX && (i >=0 && (length xs)
↪  >= 2 ))
                then True
                else False

swapiaux :: Stack -> Stack
swapiaux (x:y:ys) = (y:x:ys)


cola :: [a] -> [a]
cola [] = []
cola (_:xs) = xs

cabe :: [a] -> a
cabe [] = error "List vacia"
cabe (x:xs) = x

ngetaux :: Stack -> Stack
ngetaux [] = []
ngetaux (x:xs) = if ( (isNat x) && (   (  1<=(comToNat x) &&
↪  ( (comToNat x) <= length( ( xs )    ) &&  (isNat((x:xs)
↪  !! (comToNat x) ) ) ) )     )    )
              then  [((x:xs) !! (comToNat x))] ++ xs
              else xs

validaStack :: Stack -> Command -> Bool
validaStack s com = case com of
                      POP   -> if (length s ) > 0
                               then True
                               else False
                      SWAP  -> if (length s) > 1
                               then True
                               else False
                      SEL   -> if (length s) > 2
                               then True
```

```
                                    else False
                              NGET  -> if(isNat (cabe s))
                                    then if((length s)-1 >=
                                    ↪  comToNat(cabe s))
                                         then True
                                         else False
                                    else False

execaux :: Stack -> Stack
execaux [] = []
execaux (x:xs) = validExec x

validExec :: Command -> Stack
validExec (ES xs) = xs
validExec _ = error "Incompatible"

litS :: Stack -> Bool
litS [] = True
litS (x:xs) = if isNat x
              then True && litS xs
              else False
```

Ejercicio Semanal 2

En este semanal vimos el lenguaje EAB. Importamos `Data.List` para operaciones con listas.

0.4. Lenguaje EAB

```
data Exp = V Identifier | I Int | B Bool
| Add Exp Exp | Mul Exp Exp | Succ Exp | Pred Exp
| Not Exp | And Exp Exp | Or Exp Exp
| Lt Exp Exp | Gt Exp Exp | Eq Exp Exp
| If Exp Exp Exp
| Let Identifier Exp Exp deriving (Eq, Show)
```

Se creó una instancia de la clase Show, la cual omitiremos por espacio. Tenemos los siguientes tipos para representar identificadores y tuplas de identificadores con expresiones.

```
type Substitution = (Identifier, Exp)
type Identifier = String
```

Importamos para funciones con listas.

```
import Data.List
```

Un ejemplo de una EAB.

```
(Let ``x'' (I 1) (V ``x''))
```

Las siguientes son funciones recursivas para el tipo del lenguaje EAB.

```
-- | frVars. Función que obtiene el conjunto de variables
-- |         libres de una expresion.
frVars :: Exp -> [Identifier]
frVars (V x) = [x]
frVars (I _) = []
frVars (B _) = []
frVars (Add a b) = frVars a `union` frVars b
frVars (Mul a b) = frVars a `union` frVars b
frVars (Succ x) = frVars x
frVars (Pred x ) = frVars x
frVars (Not x) = frVars x
frVars (And p q) = frVars p `union` frVars q
frVars (Or p q) = frVars p `union` frVars q
frVars (Lt a b) = frVars a `union` frVars b
frVars (Gt a b) = frVars a `union` frVars b
frVars (Eq a b) = frVars a `union` frVars b
frVars (If b p q) = frVars b `union` frVars p `union` frVars
↪  q
frVars (Let x p q) = (frVars p `union` frVars q) \\ [x]

-- | subst. Función que aplica la substitucion a la
-- |        expresión dada en caso de ser posible.
subst :: Exp -> Substitution -> Exp
subst (V x) (y, e) = if (x == y) then e else V x
subst (I n) _ = (I n)
subst (B b) _ = (B b)
subst (Add a b) s = Add(subst a s)(subst b s)
subst (Mul a b) s = Mul(subst a s)(subst b s)
subst (Succ x) s = Succ(subst x s)
subst (Pred x) s = Pred(subst x s)
subst (Not x) s = Not(subst x s)
subst (And p q) s = And(subst p s)(subst q s)
subst (Or p q) s = Or(subst p s)(subst q s)
subst (Lt a b) s = Lt(subst a s)(subst b s)
subst (Gt a b) s = Gt(subst a s)(subst b s)
```

```
subst (Eq a b) s = Eq(subst a s)(subst b s)
subst (If b p q) s = If(subst b s)(subst p s)(subst q s)
subst (Let x e1 e2) (y,e) = if(elem x ([y] ++  frVars e))
                     then error "Could not apply the
                      ↪  substitucion"
                     else Let x (subst e1 (y,e)) (subst e2
                      ↪  (y,e))

-- | alphaEq. Función que determina si dos expresiones son
-- |          alfa-equivalentes.
alphaEq :: Exp -> Exp -> Bool
alphaEq :: Exp -> Exp -> Bool
alphaEq (V x) (V y) = x == y
alphaEq (I x) (I y) = x == y
alphaEq (B x) (B y) = x == y
alphaEq (Add a1 a2) (Add b1 b2) = (alphaEq a1 b1) &&
 ↪  (alphaEq a2 b2)
alphaEq (Mul a1 a2) (Mul b1 b2) = (alphaEq a1 b1) &&
 ↪  (alphaEq a2 b2)
alphaEq (Succ a1) (Succ b1) = (alphaEq a1 b1)
alphaEq (Pred a1) (Pred b1) = (alphaEq a1 b1)
alphaEq (Not a1) (Not b1) = (alphaEq a1 b1)
alphaEq (And a1 a2) (And b1 b2) = (alphaEq a1 b1) &&
 ↪  (alphaEq a2 b2)
alphaEq (Or a1 a2) (Or b1 b2) = (alphaEq a1 b1) && (alphaEq
 ↪  a2 b2)
alphaEq (Lt a1 a2) (Lt b1 b2) = (alphaEq a1 b1) && (alphaEq
 ↪  a2 b2)
alphaEq (Gt a1 a2) (Gt b1 b2) = (alphaEq a1 b1) && (alphaEq
 ↪  a2 b2)
alphaEq (Eq a1 a2) (Eq b1 b2) = (alphaEq a1 b1) && (alphaEq
 ↪  a2 b2)
alphaEq (If b t f) (If b1 t1 f1) = (alphaEq b1 b) &&
 ↪  (alphaEq t t1) && (alphaEq f f1)
alphaEq (Let x a1 a2) (Let y b1 b2) = alphaEq a1 b1 &&
 ↪  alphaEq (subst a2 (x,a1)) (subst b2 (y, b1))
alphaEq _ _ = False
```

Práctica 2

0.5. Semántica dinámica EAB

En esta practica podemos poner en practica el tema de semántica visto en clase. Se usara la semántica operacional para definir el comportamiento de los programas a través de un sistema de transiciones.
Utilizamos la definición del leguaje de expresiones aritmetico booleanas (EAB).
Las siguientes son funciones para la implementación de la semántica dinámica.

```
-- | eval1. Función que devuelve la transición tal que
-- |        eval1 e = e' syss e -> e'.
eval1 :: Exp -> Exp
eval1 (V _) = error "Ya es una expresion bloqueada."
eval1 (I _) = error "Ya es una expresion bloqueada."
eval1 (B _) = error "Ya es una expresion bloqueada."
eval1 (Add a b) = if (isNat a && isNat b)
                  then I(tomaNat a + tomaNat b)
                  else if (isNat a)
                       then (Add a (eval1 b))
                       else (Add (eval1 a) b)
eval1 (Mul a b) = if (isNat a && isNat b)
                  then I(tomaNat a * tomaNat b)
                  else if (isNat a)
                       then (Mul a (eval1 b))
                       else (Mul (eval1 a) b)
eval1 (Succ a) = if(isNat a)
```

```
                 then I(tomaNat a + 1)
                 else Succ(eval1 a)
eval1 (Pred a) = if(isNat a)
                 then I(tomaNat a - 1)
                 else Pred(eval1 a)
eval1 (Not x) = if(esBool x)
                then B (not (tomaBool x))
                else (Not (eval1 x))
eval1 (And a b) = if(esBool a && esBool b)
                  then B(tomaBool a && tomaBool b)
                  else if (esBool a)
                      then And a (eval1 b)
                      else And (eval1 a) b
eval1 (Or a b) = if(esBool a && esBool b)
                 then B(tomaBool a || tomaBool b)
                 else if (esBool a)
                      then Or a (eval1 b)
                      else Or (eval1 a) b
eval1 (Lt a b) = if(isNat a && isNat b)
                 then B(tomaNat b < tomaNat a)
                 else if(isNat a)
                      then Lt a (eval1 b)
                      else Lt (eval1 a) b
eval1 (Gt a b) = if(isNat a && isNat b)
                 then B(tomaNat a < tomaNat b)
                 else if(isNat a)
                      then Gt a (eval1 b)
                      else Gt (eval1 a) b
eval1 (Eq a b) = if(isNat a && isNat b)
                 then B(tomaNat b == tomaNat a)
                 else if(isNat a)
                      then Eq a (eval1 b)
                      else Eq (eval1 a) b
eval1 (If b t f) = if(esBool b)
                   then if(tomaBool b && True)
                        then t
                        else f
                   else If (eval1 b) t f
eval1 (Let x a b) = if(block a)
                    then subst b (x, a)
```

```
                    else  (Let x (eval1 a) b)
-- | evals. Función que devuelve la transicion tal que
-- |        evals e = e' syss e
-- |        ->* e' y e' esta bloqueado.
evals :: Exp -> Exp
evals (V x) = V x
evals (I n) = I n
evals (B b) = B b
evals (Add (I n) (I m)) = I (n + m)
evals (Add (I n) e2) = if(block e2)
                       then (Add (I n) e2)
                       else evals(Add (I n) (evals(e2)))
evals (Add e1 e2) = if (block e1)
                    then (Add e1 e2)
                    else evals(Add (evals e1) e2)
evals (Mul (I n)(I m)) = I (n * m)
evals (Mul (I n) e2) = if(block e2)
                       then (Mul (I n) e2)
                       else evals(Mul (I n) (evals(e2)))
evals (Mul e1 e2) = if(block e1)
                    then (Mul e1 e2)
                    else evals(Mul (evals e1) e2)
evals (Succ (I n)) = I (n + 1)
evals (Succ e1) = if(block e1)
                  then (Succ e1)
                  else evals(Succ (evals e1))
evals (Pred (I n)) = I (n - 1)
evals (Pred e1) = if(block e1)
                  then (Pred e1)
                  else evals(Pred (evals e1))
evals (Not (B b)) = B (not b)
evals (Not x) = if(block x)
                then (Not x)
                else evals(Not (evals x))
evals (And (B p) (B q)) = B(p && q)
evals (And (B p) e2) = if(block e2)
                       then (And (B p) e2)
                       else evals(And (B p) (evals(e2)))
evals (And e1 e2) = if(block e1)
                    then (And e1 e2)
```

```
                        else evals(And (evals(e1)) e2)
evals (Or (B p) (B q)) = B(p || q)
evals (Or (B p) e2) = if(block e2)
                          then (Or (B p) e2)
                          else evals(Or (B p) (evals(e2)))
evals (Or e1 e2) = if(block e1)
                       then (Or e1 e2)
                       else evals(Or (evals e1) e2)
evals (Lt (I a) (I b)) = B(b < a)
evals (Lt (I a) e2) = if(block e2)
                          then (Lt (I a) e2)
                          else evals(Lt (I a) (evals(e2)))
evals (Lt e1 e2) = if(block e1)
                       then (Lt e1 e2)
                       else evals(Lt (evals(e1)) e2)
evals (Gt (I a) (I b)) = B(a < b)
evals (Gt (I a) e2) = if(block e2)
                          then (Gt (I a) e2)
                          else evals(Gt (I a) (evals(e2)))
evals (Gt e1 e2) = if(block e1)
                       then (Gt e1 e2)
                       else evals(Gt (evals(e1)) e2)
evals (Eq (I a) (I b)) = B(a == b)
evals (Eq (I a) e2) = if(block e2)
                          then (Eq (I a) e2)
                          else evals(Eq (I a) (evals(e2)))
evals (Eq e1 e2) = if(block e1)
                       then (Eq e1 e2)
                       else evals(Eq (evals e1) e2)
evals (If (B True) e1 _) = e1
evals (If (B False) _ e2) = e2
evals (If b e1 e2) = if(block b)
                         then (If b e1 e2)
                         else evals(If (evals b) e1 e2)
evals (Let x (I n) c) = evals(subst c (x, (I n)))
evals (Let x (B b) c) = evals(subst c (x, (B b)))
evals (Let x a c) = if(block a)
                        then evals(subst c (x, a))
                        else evals(Let x (evals a) c)
```

```haskell
-- | eval. Funcion que devuelve la evaluación de un programa
-- |       tal que eval e=e' syss e ->* e' y e' es un valor.
-- |       En caso de que e' no sea un valor deberá mostrar
-- |       mensaje de error particular del operador que lo
-- |       causó.
eval :: Exp -> Exp
eval (V x) = V x
eval (I n) = I n
eval (B b) = B b
eval (Add a b) = let
  x = evals(Add a b)
  in
    if(isNat(x))
    then x
    else error "[Add] Expects two Nat."
eval (Mul a b) = let
  x = evals(Mul a b)
  in
    if(isNat x)
    then x
    else error "[Mul] Expects two Nat."
eval (Succ a) = let
  x = evals(Succ a)
  in
    if(isNat x)
    then x
    else error "[Succ] Expects one Nat."
eval (Pred a) = let
  x = evals(Pred a)
  in
    if(isNat x)
    then x
    else error "[Pred] Expects one Nat."
eval (Not x) = let
  x = evals(Not x)
  in
    if(esBool x)
    then x
    else error "[Not] Expects one Boolean."
eval (And a b) = let
```

```
    x = evals(And a b)
    in
      if(esBool x)
      then x
      else error "[And] Expects two Boolean."
  eval (Or a b) = let
    x = evals(Or a b)
    in
      if(esBool x)
      then x
      else error "[Or] Expects two Boolean."
  eval (Lt a b) = let
    x = evals(Lt a b)
    in
      if(esBool x)
      then x
      else error "[Lt] Expects two Nat."
  eval (Gt a b) = let
    x = evals(Gt a b)
    in
      if(esBool x)
      then x
      else error "[Gt] Expects two Nat."
  eval (Eq a b) = let
    x = evals(Eq a b)
    in
      if(esBool x)
      then x
      else error "[Eq] Expects two Nat."
  eval (If b a c) = let
    x = evals(If b a c)
    in
      if x == a || x == b
      then x
      else error "[If] Expects one Boolean and two Exp."
  eval (Let y a b) = let
    x = evals(Let y a b)
    in
      if(isNat x || esBool x)
      then x
```

```
    else error "[Let] Expects one Var and two Exp."
```

0.6. Semántica estática EAB

Agregamos el siguiente tipo para la implementación de la semántica estática.

```
data Type = Nat | Boolean
```

Modelamos el contexto como una lista de pares que almacenan el nombre de las variables junto con su tipo.

```
type Decl = (Identifier, Type)

type TypCtx = [Decl]
```

La siguiente función implementa la semántica estática.

```
-- | vt. Funcion que verifica el tipado de un programa tal
-- |     que vt Γ e T = True syss Γ ⊢ e:T
vt :: TypCtxt -> Exp -> Type -> Bool
vt [] (V x) t = False
vt ((a,b):xs) (V x) t = if x == a
                        then b == t
                        else vt xs (V x) t
vt l (I n) t = t == Nat
vt l (B b) t = t == Boolean
vt l (Add e1 e2) t = t == Nat &&
                     vt l e1 t &&
                     vt l e2 t
vt l (Mul e1 e2) t = t == Nat &&
                     vt l e1 t &&
                     vt l e2 t
vt l (Succ e) t = vt l e t &&
                  t == Nat
vt l (Pred e) t = vt l e t &&
                  t == Nat
```

```
vt l (Not e) t = t == Boolean &&
                 vt l e t
vt l (And e1 e2) t = t == Boolean &&
                     vt l e1 t &&
                     vt l e2 t
vt l (Or e1 e2) t = t == Boolean &&
                     vt l e1 t &&
                     vt l e2 t
vt l (Lt e1 e2) t = t == Boolean &&
                    vt l e1 Nat &&
                    vt l e2 Nat
vt l (Gt e1 e2) t = t == Boolean &&
                    vt l e1 Nat &&
                    vt l e2 Nat
vt l (Eq e1 e2) t = t == Boolean &&
                    vt l e1 Nat &&
                    vt l e2 Nat
vt l (If b e1 e2) t = vt l b Boolean &&
                      vt l e1 t &&
                      vt l e2 t
vt l (Let id e1 e2) s =
  let
    x = evals e1
  in
    vt l e1 (getType x) &&
    vt (l++[(id, getType e1)]) e2 s
```

Las siguientes son funciones auxiliares que se usararon en la resolución de la práctica.

```
getType :: Exp -> Type
getType (I _) = Nat
getType (B _) = Boolean
getType op = case op of
  Add _ _ -> Nat
  Mul _ _ -> Nat
  Succ  _ -> Nat
  Pred  _ -> Nat
  Not   _ -> Boolean
```

```
    And _ _ -> Boolean
    Or  _ _ -> Boolean
    Lt  _ _ -> Boolean
    Gt  _ _ -> Boolean
    Eq  _ _ -> Boolean
    If _ a _-> getType a
    Let _ _ b-> getType b

-- | block. Función que nos dice si una expresion
↪  está bloqueda o no.
block :: Exp -> Bool
block (V _) = True
block (I _) = True
block (B _) = True
block (Add (I _) (I _)) = False
block (Add a b) = if(isNat (evals a) && isNat(evals b))
                  then False
                  else True
block (Mul (I _) (I _)) = False
block (Mul a b) = if(isNat(evals a) && isNat(evals b))
                  then False
                  else True
block (Succ (I _)) = False
block (Succ a) = if(isNat(evals a))
                 then False
                 else True
block (Pred (I _)) = False
block (Pred a) = if(isNat(evals a))
                 then False
                 else True
block (Not (B _)) = False
block (Not x) = if(esBool(evals x))
                then False
                else True
block (And (B _) (B _)) = False
block (And a b) = if(esBool(evals a) && esBool(evals b))
                  then False
                  else True
block (Or (B _) (B _)) = False
block (Or a b) = if(esBool(evals a) && esBool(evals b))
```

```
                then False
                else True
block (Lt (I _) (I _)) = False
block (Lt a b) = if(isNat(evals a) && isNat(evals b))
                then False
                else True
block (Gt (I _) (I _)) = False
block (Gt a b) = if(isNat(evals a) && isNat(evals b))
                then False
                else True
block (Eq (I _) (I _)) = False
block (Eq a b) = if(isNat(evals a) && isNat(evals b))
                then False
                else True
block (If (B _) _ _) = False
block (If b _ _) = if(esBool(evals b))
                    then False
                    else True
block (Let _ (I _) _) = False
block (Let _ (B _) _) = False
block (Let _ a _) = if(isNat(evals a) || esBool(evals a))
↪  --Mmmm no lo sé Rick
                   then False
                   else True

-- | tomaBool. Función que devuelve la parte booleana de una
↪  EAB.
tomaBool :: Exp -> Bool
tomaBool (B b) = b
tomaBool _ = error "No es un booleano"

-- | esBool. Función que nos dice si una EAB es un booleano.
esBool :: Exp -> Bool
esBool (B _) = True
esBool _ = False

-- | isNat. Función que nos dice si una EAB es un natural.
isNat :: Exp -> Bool
isNat (I _) = True
isNat _ = False
```

```
-- | tomaNat. Función que devuelve la parte natural de una
↪  EAB.
tomaNat :: Exp -> Int
tomaNat (I n) = n
tomaNat _ = error "no es un número"
```

Práctica 3

0.7. Cálculo lambda sin tipos

El cálculo lambda consiste siplemente en tres términos y todas las combinaciones recursivas válidas de estos términos. Las definimos como sigue.

```
data Expr = Var Identifier
          | Lam Identifier Expr
          | App Expr Expr deriving(Eq)
```

Definimos los siguientes tipos.

```
type Identifier = String

type Substitution = ( Identifier , Expr )
```

Por ejemplo, la siguiente expresión $\lambda x.\lambda y.xy$, la representaremos en Haskell como sigue. $\backslash x \rightarrow \backslash y \rightarrow (xy)$. Todo esto representa las expresiones del calculo lambda sin tipos y el reducto para la funcion de sustitucion.
Se realizaron las siguiente funciones que representan la semántica operacional en en calculo lambda sin tipos.

```
-- | frVars. Obtiene el conjunto de variables libres de una
-- |         expresion.
frVars :: Expr -> [Identifier]
frVars (Var x) = [x]
frVars (Lam x e) = [y | y <- (frVars e) , y/=x ]
```

```
frVars (App e1 e2) = [x | x <- (frVars e1 `union` frVars
↪  e2)]

-- | lkVars. Obtiene el conjunto de variables ligadas de una
-- |         expresion.
lkVars :: Expr -> [Identifier]
lkVars (Var _) = []
lkVars (Lam x e) = [x] `union` lkVars e--[y | y <- (lkVars
↪  e), y == x]
lkVars (App e1 e2) = lkVars e1 `union` lkVars e2

-- | incrVar. Dado un identificador, si este no termina en
-- |          numero le agrega el sufijo 1, en caso
↪  contrario
-- |          toma el valor del numero y lo incrementa en 1.
incrVar :: Identifier -> Identifier
incrVar xs = if (elem (last xs) (['a'..'z']++['A'..'Z']))
             then xs ++ show 1
             else init xs ++ show((read[last xs])+1)

-- | alphaExpr. Toma una expresion lambda y devuelve una
-- |            alpha-equivalente utilizando la funcion
-- |            incrVar hasta encontrar un nombre que no
-- |            aparezca en el cuerpo.
alphaExpr :: Expr -> Expr
alphaExpr (Var x) = Var (incrVar x)
alphaExpr (Lam x e) =
  let
    nw = findId x (Lam x e)
  in
    alphaAux (Lam x e) x nw
alphaExpr (App e1 e2) = App (alphaExpr e1) (alphaExpr e2)

-- | subst. Aplica la sustitucion a la expresion dada.
subst :: Expr -> Substitution -> Expr
subst (Var v) (i,s)
                | v == i = s
                | otherwise = (Var v)
subst (Lam x e) (i,s)
                | x == i = (Lam x e)
```

```
                  | x `elem` (frVars s) = Lam (incrVar x)
                  ↪  (subst e (i,s))
                  | otherwise = Lam x (subst e (i,s))
subst (App e e1) (i,s) = App (subst e (i,s)) (subst e1
↪  (i,s))

-- | beta. Aplica un paso de la beta reduccion.
beta :: Expr -> Expr
beta (App (Lam x t) y) = (subst (t) (x, y))
beta (Lam x e) = Lam x (beta e)
beta (App e1 e2) = App e1 (beta e2)
beta (Var x) = Var x

-- | locked. Determina si una expresion esta bloqueada, es
-- |         decir, no se pueden hacer mas reducciones.
locked :: Expr -> Bool
locked (Var x) = True
locked (App e1 e2) = locked e1 && locked e2
locked (App (Lam x e1) e2) = False
locked (Lam x e) = locked e

-- | eval. Evalua una expresion lambda aplicando beta
-- |       reducciones hasta quedar bloqueada.
eval :: Expr -> Expr
eval x = if x == (evalaux x) then x else beta (evalaux x)
```

Las siguientes son funciones auxiliares que se usaron en la resolución de la práctica.

```
-- | alphaAux. Función que dada una expresión y un tipo
↪  'OtroSubs' devuelve
-- |           la expresión equivalente a el segundo
↪  elemento de 'OtroSubs'.
alphaAux :: Expr -> Identifier -> Identifier -> Expr
alphaAux (Var x) x1 x2 = if (x == x1) then Var x2 else Var x
alphaAux (Lam x e) x1 x2 = if x == x1 then Lam x2 (alphaAux
↪  e x1 x2) else Lam x (alphaAux e x1 x2)
alphaAux (App e1 e2) x1 x2 = App (alphaAux e1 x1 x2)
↪  (alphaAux e2 x1 x2)
```

```
-- | findId. Función que devuelve un 'Identifier' que no se
↪  encuentre en las
-- |         variables de la expresión dada.
findId :: Identifier -> Expr -> Identifier
findId x e =
  let
    x1 = incrVar x
  in
    if x1 `elem` lkVars (Lam x e) || x1 `elem` frVars (Lam x
    ↪  e)
    then findId x1 (Lam x e)
    else x1

-- | evalaux. Función recursiva auxiliar que evalúa una
↪  función del cálculo lambda.
evalaux :: Expr -> Expr
evalaux (App (Lam v e) d) = eval (subst e (v, d))
evalaux (App e f) = App (eval e) (eval f)
evalaux (Lam x e) = Lam x (eval e)
evalaux e = e
```

Práctica 4

0.8. Algoritmo W

W es un algoritmo para inferir tipos basados en sus usos. Formaliza la intuición de que un tipo puede ser inferido por la acción que realiza.
La práctica consiste en el desarrollo de dicho algoritmo.
Se implementarán las siguientes funciones:

```haskell
-- |The 'erase' function transforms a MinHs expression to a
↪  UMinHs expression.
erase :: MH.Expr -> UMH.Expr
erase e = case e of
  MH.V x -> UMH.V x
  MH.I n -> UMH.I n
  MH.B b -> UMH.B b
  MH.Add e1 e2 -> UMH.Add (erase e1)(erase e2)
  MH.Mul e1 e2 -> UMH.Mul (erase e1)(erase e2)
  MH.Succ e1 -> UMH.Succ (erase e1)
  MH.Pred e1 -> UMH.Pred (erase e1)
  MH.And e1 e2 -> UMH.And (erase e1)(erase e2)
  MH.Or e1 e2 -> UMH.Or (erase e1)(erase e2)
  MH.Not e1 -> UMH.Not (erase e1)
  MH.Lt e1 e2 -> UMH.Lt (erase e1)(erase e2)
  MH.Gt e1 e2 -> UMH.Gt (erase e1)(erase e2)
  MH.Eq e1 e2 -> UMH.Eq (erase e1)(erase e2)
  MH.If b e1 e2 -> UMH.If (erase b)(erase e1)(erase e2)
  MH.Let x _ e1 e2 -> UMH.Let x(erase e1)(erase e2)
```

```
    MH.Fun x _ e1 -> UMH.Fun x (erase e1)
    MH.FunF n x _ _ e1 -> UMH.FunF n x (erase e1)
    MH.App e1 e2 -> UMH.App (erase e1)(erase e2)

  -- |The 'newVType' function returns a new variable type that
  ↪  is not contained in the given set.
  newVType :: [VType] -> VType
  newVType vs = case vs of
    [] -> T 1
    (T x : xs) -> if (T (x + 1)`elem` xs)
                  then newVType xs
                  else T(x + 1)
    (_:xs) -> newVType xs

  w :: UMH.Expr -> Judgement
  --w e = error "Not yet implemented."
  w e = let (Assertion (ctx,e',t),_) = algW e [] in Assertion
  ↪  (deleteDuplicate
ctx, e',t) where
           deleteDuplicate [] = []
           deleteDuplicate (x:xs) = x : deleteDuplicate
           ↪  (filter (/= x) xs)
```

Es importante notar que la práctica solo se trata de completar dichas funciones, ya que las funciones auxiliares y los archivos como MiniHaskell y UntypedMiniHaskell, o los archivos de Syntax ya están escritos. El motivo de esta entrada es solo lo que se hizo en el laboratorio, así que explicar dichos archivos está fuera del alcance.

Ejercicio Semanal 3

En este semanal implementamos un lenguaje llamado miniC, el cual es un lenguaje imperativo que tiene como núcleo el lenguaje de expresiones aritmetico booleanas.

0.9. MiniC

Recordemos el lenguaje de EAB y agregamos el comportamiento de nuevos constructores para miniC.

```
data Exp = V identifier | I Int | B Bool
| Add Exp Exp | Mul Exp Exp | Succ Exp | Pred Exp
| And Exp Exp | Or Exp Exp  | Not Exp
| Lt Exp Exp  | Gt Exp Exp  | Eq Exp Exp
| If Exp Exp Exp
| Let Identifier Exp Exp
| L Int
| Alloc Exp
| Deref Exp
| Assig Exp Exp
| Void
| Seq
| While Exp Exp
```

Un ejemplo trivial de un programa en miniC se puede ver como sigue:

```
While (B True) Void
```

Tenemos los tipos anteriores y agregamos nuevos tipos para poder representar la memoria y direcciones.

```
type Identifier = String

type Addr = Exp

type Value = Exp

type Cell = (Addr, Value)

type Mem = [Cell]

type Substitution = (Identifier, Exp)
```

Importamos

```
import Data.List

import Data.Void
```

Las siguientes son funciones que implementamos para poder representar el lenguaje miniC.

```
-- | domain. Dada una memoria, obtiene todas sus
↪  direcciones.
domain :: Mem -> [Int]
domain m = case m of
             [] -> []
             (L l, _):xs -> [l] ++ domain xs
             (_, _):xs -> error "Corrupted memory."

-- | newL. Dada una memoria genera una nueva direccion.
newL :: Mem -> Addr
newL m = case m of
           [] -> L 0
           (L l, o):xs -> if(l+1 `elem` domain m)
                          then newL xs
```

```
                            else L (l + 1)

-- | accessM. Dada una direccion de memoria, devuelve el
-- |          valor contenido en la celda con tal direccion.
accessM :: Addr -> Mem -> Maybe Value
accessM _ [] = Nothing
accessM (L l) ((L l', o):xs) = if l == l'
                               then Just o
                               else accessM (L l) xs
accessM (L l) ((_, o):xs) = error "Corrupted memory."

-- | updateM. Dada una celda de memoria, actualiza el valor.
updateM :: Cell -> Mem -> Mem
updateM _ [] = error "Memory address does not exist."
updateM a@(L l, Void) (b@(L l', _):xs) = if l == l'
                                         then [(L l, Void)]
                                         ↪  ++ xs
                                         else [b] ++ updateM
                                         ↪  a xs
updateM a@(L l, I n) (b@(L l',  _):xs) = if l == l'
                                          then [(L l, I n)]
                                          ↪  ++ xs
                                          else [b] ++
                                          ↪  updateM a xs
updateM a@(L l, B n) (b@(L l',  _):xs) = if l == l'
                                          then [(L l, B n)]
                                          ↪  ++ xs
                                          else [b] ++
                                          ↪  updateM a xs
updateM (L _, _) ((L _, _):xs) = error "Memory can only
↪  store values."
updateM (L _, _) ((_ , _):xs) = error "Corrupted memory."

-- | frVars. Extiende esta funcion del lenguaje EAB con las
-- |         nuevas funciones.
frVars :: Exp -> [Identifier]
frVars (V x) = [x]
frVars (I _) = []
frVars (B _) = []
frVars (Add a b) = frVars a `union` frVars b
```

```
frVars (Mul a b) = frVars a `union` frVars b
frVars (Succ x) = frVars x
frVars (Pred x ) = frVars x
frVars (Not x) = frVars x
frVars (And p q) = frVars p `union` frVars q
frVars (Or p q) = frVars p `union` frVars q
frVars (Lt a b) = frVars a `union` frVars b
frVars (Gt a b) = frVars a `union` frVars b
frVars (Eq a b) = frVars a `union` frVars b
frVars (If b p q) = frVars b `union` frVars p `union` frVars
↪ q
frVars (Let x p q) = frVars p `union` [y | y <- frVars q, y
↪ /= x]
frVars (Void) = []
frVars (L _) = []
frVars (Alloc e1) = frVars e1
frVars (Deref e1) = frVars e1
frVars (Assig e1 e2) = frVars e1 `union` frVars e2
frVars (Seq e1 e2) = frVars e1 `union` frVars e2
frVars (While e1 e2) = frVars e1 `union` frVars e2

-- | subst. Extiende esta funcion del lenguaje EAB con las
-- |        nuevas funciones.
subst :: Exp -> Substitution -> Exp
subst (V x) (y, e) = if (x == y)
                     then e
                     else V x
subst (I n) _ = (I n)
subst (B b) _ = (B b)
subst (Add a b) s = Add(subst a s)(subst b s)
subst (Mul a b) s = Mul(subst a s)(subst b s)
subst (Succ x) s = Succ(subst x s)
subst (Pred x) s = Pred(subst x s)
subst (Not x) s = Not(subst x s)
subst (And p q) s = And(subst p s)(subst q s)
subst (Or p q) s = Or(subst p s)(subst q s)
subst (Lt a b) s = Lt(subst a s)(subst b s)
subst (Gt a b) s = Gt(subst a s)(subst b s)
subst (Eq a b) s = Eq(subst a s)(subst b s)
subst (If b p q) s = If(subst b s)(subst p s)(subst q s)
```

```
subst (Let x e1 e2) (y,e) = if(elem x ([y] ++  frVars e))
                            then error "Could not apply the
                            ↪  substitution"
                            else Let x (subst e1 (y,e))
                            ↪  (subst e2 (y,e))
subst (Void) _ = Void
subst (L l) _ = (L l)
subst (Alloc e1) s = Alloc(subst e1 s)
subst (Deref e1) s = Deref(subst e1 s)
subst (Assig e1 e2) s = Assig(subst e1 s) (subst e2 s)
subst (Seq e1 e2) s = Seq(subst e1 s) (subst e2 s)
subst (While e1 e2) s = While(subst e1 s) (subst e2 s)

-- | eval1. Extiende esta funcion del lenguaje EAB con las
-- |        nuevas funciones.
eval1 :: (Mem, Exp) -> (Mem, Exp)
eval1 (m, e) = case e of
                 (V v) -> error "Ya es una expresión
                 ↪  bloqueada"
                 (I n) -> error "Ya es una expresión
                 ↪  bloqueada"
                 (B b) -> error "Ya es una expresión
                 ↪  bloqueda"
                 (Add e1 e2) -> if (isNat(m, e1) && isNat(m,
                 ↪  e1))
                               then (m, I(tomaNat e1 +
                               ↪  tomaNat e2))
                               else if (isNat(m, e1))
                                    then let (m', e') =
                                    ↪  eval1 (m, e2)
                                         in
                                           (m', (Add e1 e'))
                                    else let (m', e') =
                                    ↪  eval1 (m, e1)
                                         in
                                           (m', (Add e' e2))
                 (Mul e1 e2) -> if (isNat(m, e1) && isNat(m,
                 ↪  e1))
                                then (m, I(tomaNat e1 *
                                ↪  tomaNat e2))
```

```
                   else if (isNat(m, e1))
                        then let (m', e') =
                        ↪  eval1 (m, e2)
                             in
                               (m', (Mul e1
                               ↪  e'))
                        else let (m', e') =
                        ↪  eval1 (m, e1)
                             in
                               (m', (Mul e'
                               ↪  e2))
(Succ e1) -> if (isNat(m, e1))
             then (m, I(tomaNat e1 + 1))
             else let (m', e') = eval1 (m,
             ↪  e1)
                  in
                    (m', Succ e')
(Pred e1) -> if (isNat(m, e1))
            then (m, I(tomaNat e1 - 1))
            else let (m', e') = eval1 (m,
            ↪  e1)
                 in
                   (m', Pred e')
(Not e1) -> if (esBool(m, e1))
            then (m, B(not (tomaBool e1)))
            else let (m', e') = eval1 (m,
            ↪  e1)
                 in
                   (m', Not e')
(And e1 e2) -> if (esBool(m, e1) &&
↪  esBool(m, e2))
               then (m, B(tomaBool e1 &&
               ↪  tomaBool e2))
               else if (esBool(m, e1))
                    then let (m', e') =
                    ↪  eval1 (m, e2)
                         in
                           (m', (And e1
                           ↪  e'))
```

```
                        else let (m', e') =
                        ↪  eval1 (m, e1)
                             in
                               (m', (And e'
                               ↪  e2))
(Or e1 e2) -> if (esBool(m, e1) &&
↪  esBool(m, e2))
              then (m, B(tomaBool e1 ||
              ↪  tomaBool e2))
              else if (esBool(m, e1))
  then let (m', e') = eval1 (m, e2)
       in
         (m', (Or e1 e'))
                   else let (m', e') =
                   ↪  eval1 (m, e1)
                        in
                          (m', (Or e' e2))
(Lt e1 e2) -> if (isNat(m, e1) && isNat(m,
↪  e1))
              then (m, B(tomaNat e2 <
              ↪  tomaNat e1))
              else if (isNat(m, e1))
                   then let (m', e') =
                   ↪  eval1 (m, e2)
                        in
                          (m', (Lt e1 e'))
                   else let (m', e') =
                   ↪  eval1 (m, e1)
                        in
                          (m', (Lt e' e2))
(Gt e1 e2) -> if (isNat(m, e1) && isNat(m,
↪  e1))
              then (m, B(tomaNat e1 <
              ↪  tomaNat e2))
              else if (isNat(m, e1))
                   then let (m', e') =
                   ↪  eval1 (m, e2)
                        in
                          (m', (Gt e1 e'))
```

```
                    else let (m', e') =
                    ↪  eval1 (m, e1)
                         in
                           (m', (Gt e' e2))
(Eq e1 e2) -> if (isNat(m, e1) && isNat(m,
↪  e1))
              then (m, B(tomaNat e1 ==
              ↪  tomaNat e2))
              else if (isNat(m, e1))
                   then let (m', e') =
                   ↪  eval1 (m, e2)
                        in
                          (m', (Eq e1 e'))
                   else let (m', e') =
                   ↪  eval1 (m, e1)
                        in
                          (m', (Eq e' e2))
(If b e1 e2) -> if(esBool (m, b))
                then if(tomaBool b && True)
                     then (m, e1)
                     else (m, e2)
                else let (m', b') = eval1
                ↪  (m, b)
                     in
                       (m', If b' e1 e2)
(Let x e1 e2) -> if(block (m, e1))
                 then (m, subst e2 (x, e1))
                 else let (m', e') = eval1
                 ↪  (m, e1)
                      in
                        (m', Let x e' e2)
(Void) -> error "Ya es una expresión
↪  bloqueda."
(L _) -> error "Esto no debería estar
↪  aquí."
(Alloc e1) -> if(block (m, e1))
              then (((newL m),e1):m, newL
              ↪  m)
              else let (m', e') = eval1 (m,
              ↪  e1)
```

```
                               in
                                 (m', Alloc e1)
                (Deref l@(L _)) -> case accessM l m of
                                   Nothing -> error
                                    ↪  "Memory address
                                    ↪  does not exist"
                                   Just v -> (m,v)
                (Deref e1) -> let (m',e') = eval1 (m,e1) in
                ↪  (m', Deref e')
                (Assig e1 e2) -> if(block (m, e1) && block
                ↪  (m, e2))
                                 then (updateM (e1,e2) m,
                                 ↪  Void)
                                 else let (m', e') = eval1
                                 ↪  (m, e1)
                                      in
                                        (m', Assig e' e2)
                (Seq Void e2) -> (m, e2)
                (Seq e1 e2) -> let (m',e') = eval1 (m, e1)
                               in
                                 (m', Seq e' e2)
                w@(While e1 e2) -> (m, If (e1) (Seq e2 w)
                ↪  Void)

-- | evals. Extiende esta funcion del lenguaje EAB con las
-- |        nuevas funciones.
evals :: (Mem, Exp) -> (Mem, Exp)
evals (m, e) = case e of
                (V v) -> (m, V v)
                (I n) -> (m, I n)
                (B b) -> (m, B b)
                (Add (I n1) (I n2)) -> (m, I(n1 + n2))
                (Add (I n1) e2) -> if (block (m,e2))
                                   then (m, Add (I n1) e2)
                                   else let (m', e') =
                                   ↪  evals (m, e2)
                                        in
                                          evals(m', Add (I
                                          ↪  n1) e')
                (Add e1 e2) -> if(block (m, e1))
```

```
                 then (m, Add e1 e2)
                 else let (m',e') = evals(m,
                 ↪  e1)
                      in
                        evals(m', Add e' e2 )
(Mul (I n1) (I n2)) -> (m, I(n1 * n2))
(Mul (I n1) e2) -> if (block (m,e2))
                     then (m, Mul (I n1) e2)
                     else let (m', e') =
                     ↪  evals (m, e2)
                          in
                            evals(m', Mul (I
                            ↪  n1) e')
(Mul e1 e2) -> if(block (m, e1))
                 then (m, Mul e1 e2)
                 else let (m',e') = evals(m,
                 ↪  e1)
                      in
                        evals(m', Mul e' e2 )
(Succ (I n)) -> (m, I(n + 1))
(Succ e1) -> if(block (m, e1))
               then (m, Succ e1)
               else let (m', e') = evals (m,
               ↪  e1)
                    in
                      (m', e')
(Pred (I n)) -> (m, I(n - 1))
(Pred e1) -> if(block (m, e1))
               then (m, Pred e1)
               else let (m', e') = evals (m,
               ↪  e1)
                    in
                      (m', e')
(Not (B b)) -> (m, B(not b))
(Not e1) -> if(block (m, e))
              then (m, Not e1)
              else let (m', e') = evals(m,
              ↪  e1)
                   in
                     evals(m', Not e')
```

```
(And (B n1) (B n2)) -> (m, B(n1 && n2))
(And (B n1) e2) -> if (block (m,e2))
                   then (m, And (B n1) e2)
                   else let (m', e') =
                   ↪  evals (m, e2)
                        in
                          evals(m', And (B
                          ↪  n1) e')
(And e1 e2) -> if(block (m, e1))
               then (m, And e1 e2)
               else let (m',e') = evals(m,
               ↪  e1)
                    in
                      evals(m', And e' e2 )
(Or (B n1) (B n2)) -> (m, B(n1 || n2))
(Or (B n1) e2) -> if (block (m,e2))
                  then (m, Or (B n1) e2)
                  else let (m', e') =
                  ↪  evals (m, e2)
                       in
                         evals(m', Or (B
                         ↪  n1) e')
(Or e1 e2) -> if(block (m, e1))
              then (m, Or e1 e2)
              else let (m',e') = evals(m,
              ↪  e1)
                   in
                     evals(m', Or e' e2 )
(Lt (I a) (I b)) -> (m, B(b < a))
(Lt (I a) e2) -> if(block (m, e2))
                 then (m, Lt (I a) e2)
                 else let (m', e') =
                 ↪  evals(m, e2)
                      in
                        evals(m', Lt (I a)
                        ↪  e')
(Lt e1 e2) -> if(block (m, e1))
              then (m, Lt e1 e2)
              else let (m', e') = evals(m,
              ↪  e1)
```

```
                    in
                      (m', Lt e' e2)
(Gt (I a) (I b)) -> (m, B(a < b))
(Gt (I a) e2) -> if(block (m, e2))
                 then (m, Gt (I a) e2)
                 else let (m', e') =
                 ↪  evals(m, e2)
                      in
                        evals(m', Gt (I a)
                         ↪  e')
(Gt e1 e2) -> if(block (m, e1))
              then (m, Gt e1 e2)
              else let (m', e') = evals(m,
              ↪  e1)
                   in
                     (m', Gt e' e2)
(Eq (I a) (I b)) -> (m, B(a == b))
(Eq (I a) e2) -> if(block (m, e2))
                 then (m, Eq (I a) e2)
                 else let (m', e') =
                 ↪  evals(m, e2)
                      in
                        evals(m', Eq (I a)
                         ↪  e')
(Eq e1 e2) -> if(block (m, e1))
              then (m, Eq e1 e2)
              else let (m', e') = evals(m,
              ↪  e1)
                   in
                     (m', Gt e' e2)
(If (B True) e1 _) -> (m, e1)
(If (B False) _ e2) -> (m, e2)
i@(If b e1 e2) -> if(block (m, b))
                  then (m, i)
                  else let (m', b') = evals
                  ↪  (m, b)
                       in
                         evals(m', If b' e1
                          ↪  e2)
```

```
(Let x (I n) c) -> evals(m, subst c (x, I
↪  n))
(Let x (B b) c) -> evals(m, subst c (x, B
↪  b))
(Let x a c) -> if(block(m, a))
               then evals(m, subst c(x,a))
               else let (m', a') = evals(m,
                ↪  a)
                    in
                      evals(m', Let x a' c)
(Void) -> (m, Void)
(L _) -> error "Esto no debería estar
↪  aquí."
(Alloc e1) -> if(block (m, e1))
              then (((newL m),e1):m, newL
               ↪  m)
              else let (m', e') = evals (m,
               ↪  e1)
                   in
                     (m', Alloc e')
(Deref l@(L _)) -> case accessM l m of
                     Nothing -> error
                      ↪  "Memory address
                      ↪  does not exist"
                     Just v -> (m,v)
(Deref e1) -> let (m',e') = evals (m,e1) in
↪  (m', Deref e')
(Assig (L l) e1) -> if(block (m, e1))
                    then ((updateM ((L
                    ↪  l),e1) m),Void)
                    else let (m', e') =
                    ↪  evals(m, e1)
                         in
                           evals(m', Assig
                            ↪  (L l) e')
(Assig e1 e2) -> let (m', e') = evals(m,
↪  e1)
                 in
                   evals(m', Assig e' e2)
(Seq Void e2) -> (m, e2)
```

```
                    (Seq e1 e2) -> let (m',e') = evals (m, e1)
                                   in
                                     evals(m', Seq e' e2)
                   w@(While e1 e2) -> evals(m, If (e1) (Seq e2
                    ↪  w) Void)

-- | eval. Extiende esta funcion del lenguaje EAB con las
-- |        nuevas funciones.
eval :: Exp -> Exp
eval :: Exp -> Exp
eval e = case evals ([],e) of
            (_, I n) -> I n
            (_, B b) -> B b
            (_, V _) -> error "[Var]"
            (_, Add _ _) -> error "[Add] Expects two Nat."
            (_, Mul _ _) -> error "[Mul] Expects two Nat."
            (_, Succ _) -> error "[Succ] Expects one Nat."
            (_, Pred _) -> error "[Pred] Expects one Nat."
            (_, Not _) -> error "[Not] Expects one Boolean."
            (_, And _ _) -> error "[And] Expects two
            ↪  Boolean."
            (_, Or _ _) -> error "[Or] Expects two Boolean."
            (_, Lt _ _) -> error "[Lt] Expects two Nat."
            (_, Gt _ _) -> error "[Gt] Expects two Nat."
            (_, Eq _ _) -> error "[Eq] Expects two Nat."
            (_, If _ _ _) -> error "[If] Expects one Boolean
            ↪  and two Exp."
            (_, Let _ _ _) -> error "[Let] Expects one Var
            ↪  and two Exp."
            (_, Void) -> Void
            (_, L l) -> L l
            (_, Alloc _) -> error "[Alloc] Expects one Exp."
            (_, Deref _) -> error "[Deref] Expects one Exp."
            (_, Assig _ _) -> error "[Deref] Expects one
            ↪  location and one Exp."
            (_, Seq _ _) -> error "[Seq] Expects two Exp."
            (_, While _ _) -> error "[While] Expects one
            ↪  condition and one Exp."
```

Ejercicio Semanal 4

En este semanal implementamos máquinas K para evaluar expresiones EAB.

0.10. máquina K

Recordemos el lenguaje de EAB y agregamos el tipo Error. También agregamos nuevos datos, para poder regresentar las evaluaciones de los marcos, al cual llamamos Frame.

```
data Exp = V identifier | I Int | B Bool
| Add Exp Exp | Mul Exp Exp | Succ Exp | Pred Exp
| And Exp Exp | Or Exp Exp  | Not Exp
| Lt Exp Exp  | Gt Exp Exp  | Eq Exp Exp
| If Exp Exp Exp
| Let Identifier Exp Exp
| Error

data Frame = AddL Pending Expr
| AddR Expr Pending
| MulL Pending Expr
| MulR Expr Pending
| SuccF Pending
| PredF Pending
| AndL Pending Expr
| AndR Expr Pending
| OrL Pending Expr
| OrR Expr Pending
```

```
  | NotF Pending
  | LtL Pending Expr
  | LtR Expr Pending
  | GtL Pending Expr
  | GtR Expr Pending
  | EqL Pending Expr
  | EqR Expr Pending
  | IfF Pending Expr Expr
  | LetF Identifier Pending Expr
```

Importamos

```
import Data.List
```

Añadimos los tipos

```
type Identifier = String

type Substitution = (Identifier, Expr)

type Pending = ()

type Stack = [Frame]
```

Usamos el siguiente `data` para representar cuando entran y salen expresiones a la máquina K.

```
data State = E(Stack, Expr)
           | R(Stack, Expr)
```

Un ejemplo de un programa en una máquina K es el siguiente, el cuál regresa un error.

```
eval (Not (I 3))
```

Agregamos un nuevo tipo para poder representar un marco, y su evaluación.

```
type Pending ()
```

Modelamos los estados de una máquina K como sigue.

```
State = E (Stack, Expr) | R (Stack, Expr)
```

Las siguientes son funciones que implementamos para poder representar las máquinas K.

```
-- | frVars. Obtiene el conjunto de variables libres de una
-- |         expresion.
frVars :: Exp -> [Identifier]
frVars (V x) = [x]
frVars (I _) = []
frVars (B _) = []
frVars (Add a b) = frVars a `union` frVars b
frVars (Mul a b) = frVars a `union` frVars b
frVars (Succ x) = frVars x
frVars (Pred x ) = frVars x
frVars (Not x) = frVars x
frVars (And p q) = frVars p `union` frVars q
frVars (Or p q) = frVars p `union` frVars q
frVars (Lt a b) = frVars a `union` frVars b
frVars (Gt a b) = frVars a `union` frVars b
frVars (Eq a b) = frVars a `union` frVars b
frVars (If b p q) = frVars b `union` frVars p `union` frVars
↪  q
frVars (Let x p q) = frVars p `union` ([y | y <- frVars q, y
↪  /= x])
frVars (Error) = []

-- | subst. Realiza la substitucion de una expresion de EAB.
subst :: Exp -> Substitution -> Exp
subst (V x) (y, e) = if (x == y)
                     then e
                     else V x
subst (I n) _ = (I n)
subst (B b) _ = (B b)
subst (Add a b) s = Add(subst a s)(subst b s)
```

```
subst (Mul a b) s = Mul(subst a s)(subst b s)
subst (Succ x) s = Succ(subst x s)
subst (Pred x) s = Pred(subst x s)
subst (Not x) s = Not(subst x s)
subst (And p q) s = And(subst p s)(subst q s)
subst (Or p q) s = Or(subst p s)(subst q s)
subst (Lt a b) s = Lt(subst a s)(subst b s)
subst (Gt a b) s = Gt(subst a s)(subst b s)
subst (Eq a b) s = Eq(subst a s)(subst b s)
subst (If b p q) s = If(subst b s)(subst p s)(subst q s)
subst (Let x e1 e2) (y,e) = if(elem x ([y] ++  frVars e))
                            then error "Could not apply the
                            ↪  substitution"
                            else Let x (subst e1 (y,e))
                            ↪  (subst e2 (y,e))
subst (Error) _ = (Error)

-- | eval1. Recibe un estado de la maquina K, y devuelve un
-- |        paso de la transicion.
eval1 :: (Mem, Exp) -> (Mem, Exp)
eval1 (E (s, I n)) = (R (s, I n))
eval1 (E (s, B b)) = (R (s, B b))
eval1 (E (s, V v)) = (R (s, V v))
eval1 (E (s, Succ e1)) = (E ((SuccF ()):s, e1))
eval1 (R ((SuccF ()):s, (I n))) = (R (s, I (n + 1)))
eval1 (E (s, Pred e1)) = (E ((PredF ()):s, e1))
eval1 (R ((PredF ()):s, (I n))) = (R (s, I (n - 1)))
eval1 (E (s, Not e1)) = (E ((NotF ()):s, e1))
eval1 (R ((NotF ()):s, (B b))) = (R (s, B (not b)))
eval1 (E (s, Add e1 e2)) = (E ((AddL () e2):s, e1))
eval1 (R ((AddL () e2):s, I n)) = (E ((AddR (I n) ()):s,
↪  e2))
eval1 (R ((AddR (I n) ()):s, I n')) = (R (s, I (n + n')))
eval1 (E (s, Mul e1 e2)) = (E ((MulL () e2):s, e1))
eval1 (R ((MulL () e2):s, I n)) = (E ((MulR (I n)()):s, e2))
eval1 (R ((MulR (I n) ()):s, I n')) = (R (s, I (n * n')))
eval1 (E (s, And e1 e2)) = (E ((AndL () e2):s, e1))
eval1 (R ((AndL () e2):s, B b)) = (E ((AndR (B b) ()):s,
↪  e2))
eval1 (R ((AndR (B b) ()):s, B b')) = (R (s, B (b && b')))
```

```
eval1 (E (s, Or e1 e2)) = (E ((OrL () e2):s, e1))
eval1 (R ((OrL () e2):s, B b)) = (E ((OrR (B b) ()):s, e2))
eval1 (R ((OrR (B b) ()):s, B b')) = (R (s, B (b || b')))
eval1 (E (s, Lt e1 e2)) = (E ((LtL () e2):s, e1))
eval1 (R ((LtL () e2):s, I n)) = (E ((LtR (I n) ()):s, e2))
eval1 (R ((LtR (I n) ()):s, I n')) = (R (s, B (n < n')))
eval1 (E (s, Gt e1 e2)) = (E ((GtL () e2):s, e1))
eval1 (R ((GtL () e2):s, I n)) = (E ((GtR (I n) ()):s, e2))
eval1 (R ((GtR (I n) ()):s, I n')) = (R (s, B (n' < n)))
eval1 (E (s, Eq e1 e2)) = (E ((EqL () e2):s, e1))
eval1 (R ((EqL () e2):s, I n)) = (E ((EqR (I n) ()):s, e2))
eval1 (R ((EqR (I n) ()):s, I n')) = (R (s, B (n' == n)))
eval1 (E (s, If b e1 e2)) = (E ((IfF () e1 e2):s, b))
eval1 (R ((IfF () e1 _):s, (B True))) = (E (s, e1))
eval1 (R ((IfF () _ e2):s, (B False))) = (E (s, e2))
eval1 (E (s, Let x e1 e2)) = (E ((LetF x () e2):s, e1))
eval1 (R ((LetF x () e2):s, v)) = (E (s, subst e2 (x,v)))
eval1 _ = (E ([], Error))

-- | evals. Recibe un estado de la maquina K y devuelve un
-- |        estado derivado de evaluar varias veces hasta
-- |        obtener la pila vacia.
evals :: (Mem, Exp) -> (Mem, Exp)
evals (E ([], I n)) = (R ([], I n))
evals (E ([], B b)) = (R ([], B b))
evals (E ([], V v)) = (R ([], V v))
evals (R ([], I n)) = (R ([], I n))
evals (R ([], B b)) = (R ([], B b))
evals (R ([], V v)) = (R ([], V v))
evals (E ([], Error)) = (E ([], Error))
evals otra = evals(eval1 otra)

-- | eval. Recibe una expresion EAB, la evalua con la
↪  maquina
-- |       K, y devuelve un valor, iniciando con la pila
-- |       vacia esta devuelve un valor a la pila.
eval :: Exp -> Exp
eval e = let
  x = evals(E ([], e))
  in
```

```
    let ex = takeExpr x
    in
      if ex == Error
      then error "No se pudo evaluar."
      else ex
```

Las siguientes son funciones auxiliares que se utilizaron en la resolución de ese ejercicio semanal.

```
-- | takeExpr. Función auxiliar que obtiene el lado derecho
↪  de
-- |          una máquina K.
takeExpr :: State -> Expr
takeExpr (E (_ ,e)) = e
takeExpr (R (_ ,e)) = e
```

Práctica 5

En esta práctica se desarrollaron dos temas vistos en clase cuales son **Excepciones** y **Continuaciones**. Para los siguientes modulos vamos a importar lo mismo:

```haskell
import Data.List
```

Tendremos los mismos `type`

```haskell
type Identifier = String

type Substitution = (Identifier, Expr)

type Pending = ()

type Stack = [Frame]
```

Además de compartir los mismos `data`.

```haskell
data State = E(Stack, Expr)
           | R(Stack, Expr)

data Expr = V Identifier | I Int  | B Bool
    | Add Expr Expr | Mul Expr Expr | Succ Expr | Pred Expr
    | And Expr Expr | Or Expr Expr  | Not Expr
    | Lt Expr Expr  | Gt Expr Expr  | Eq Expr Expr
    | If Expr Expr Expr
    | Let Identifier Expr Expr
```

```haskell
data Frame = AddL Pending Expr
           | AddR Expr Pending
           | MulL Pending Expr
           | MulR Expr Pending
           | SuccF Pending
           | PredF Pending
           | AndL Pending Expr
           | AndR Expr Pending
           | OrL Pending Expr
           | OrR Expr Pending
           | NotF Pending
           | LtL Pending Expr
           | LtR Expr Pending
           | GtL Pending Expr
           | GtR Expr Pending
           | EqL Pending Expr
           | EqR Expr Pending
           | IfF Pending Expr Expr
           | LetF Identifier Pending Expr
```

Las siguientes son funciones auxiliares que se utilizaron en la resolución de ese ejercicio semanal, en los tres módulos.

```haskell
-- | takeExpr. Función auxiliar que obtiene el lado derecho
↪  de una máquina K.
takeExpr :: State -> Expr
takeExpr (E (_ ,e)) = e
takeExpr (R (_ ,e)) = e
takeExpr (U (_, e)) = e -- Depende del módulo.
```

0.11. FEAB

En módulo llamado `FEAB` implmentamos la siguiente extensión del lenguaje `EAB`. Esta es la manera más simple de manejar errores que no tienen información asociada. Añadimos a `data Expr` Lo siguiente.

```
data Expr = V Identifier | ... | ...
          | Error
          | Catch Expr Expr
```

Añadimos los siguientes tipos.

```
type Decl = (Identifier, Type)

type TypCtxt = [Decl]
```

Agregamos lo siguiente para poder completar los tipos.

```
data Type = Integer | Boolean deriving (Eq)
```

Para representar los estados, agregamos el siguiente en `data State`.

```
data State = ... | U (State, Expr)
```

Finalmente añadimos en `data Frame` lo correspondiente a las expresiones nuevas.

```
data Frame = ... | CatchL Pending Expr
```

Las siguientes son funciones recursivas para representar este tipo de errores.

```
-- | frVars. Obtiene el conjunto de variables libres de una
-- |         expresion.
frVars :: Expr -> [Identifier]
frVars (V x) = [x]
frVars (I _) = []
frVars (B _) = []
frVars (Add a b) = frVars a `union` frVars b
frVars (Mul a b) = frVars a `union` frVars b
frVars (Succ x) = frVars x
frVars (Pred x ) = frVars x
frVars (Not x) = frVars x
```

```
frVars (And p q) = frVars p `union` frVars q
frVars (Or p q) = frVars p `union` frVars q
frVars (Lt a b) = frVars a `union` frVars b
frVars (Gt a b) = frVars a `union` frVars b
frVars (Eq a b) = frVars a `union` frVars b
frVars (If b p q) = frVars b `union` frVars p `union` frVars
 ↪ q
frVars (Let x p q) = frVars p `union` ([y | y <- frVars q, y
 ↪ /= x])
frVars (Error) = []
frVars (Catch a b) = frVars a `union` frVars b --nuevo

-- | subst. Realiza la substitucion de una expresion de EAB.
subst :: Expr -> Substitution -> Expr
subst (V x) (y, e) = if (x == y)
                     then e
                     else V x
subst (I n) _ = (I n)
subst (B b) _ = (B b)
subst (Add a b) s = Add(subst a s)(subst b s)
subst (Mul a b) s = Mul(subst a s)(subst b s)
subst (Succ x) s = Succ(subst x s)
subst (Pred x) s = Pred(subst x s)
subst (Not x) s = Not(subst x s)
subst (And p q) s = And(subst p s)(subst q s)
subst (Or p q) s = Or(subst p s)(subst q s)
subst (Lt a b) s = Lt(subst a s)(subst b s)
subst (Gt a b) s = Gt(subst a s)(subst b s)
subst (Eq a b) s = Eq(subst a s)(subst b s)
subst (If b p q) s = If(subst b s)(subst p s)(subst q s)
subst (Let x e1 e2) (y,e) = if(elem x ([y] ++  frVars e))
                            then error "Could not apply the
                            ↪ substitution"
                            else Let x (subst e1 (y,e))
                            ↪  (subst e2 (y,e))
subst (Error) _ = (Error)
subst (Catch a b) s = Catch(subst a s)(subst b s)

-- | eval1. Recibe un estado de la maquina K, y devuelve un
-- |        paso de la transicion.
```

```
eval1 :: State -> State
eval1 (E (s, I n)) = (R (s, I n))
eval1 (E (s, B b)) = (R (s, B b))
eval1 (E (s, V v)) = (R (s, V v))
eval1 (E (s, Succ e1)) = (E ((SuccF ()):s, e1))
eval1 (R ((SuccF ()):s, (I n))) = (R (s, I (n + 1)))
eval1 (E (s, Pred e1)) = (E ((PredF ()):s, e1))
eval1 (R ((PredF ()):s, (I n))) = (R (s, I (n - 1)))
eval1 (E (s, Not e1)) = (E ((NotF ()):s, e1))
eval1 (R ((NotF ()):s, (B b))) = (R (s, B (not b)))
eval1 (E (s, Add e1 e2)) = (E ((AddL () e2):s, e1))
eval1 (R ((AddL () e2):s, I n)) = (E ((AddR (I n) ()):s,
↪  e2))
eval1 (R ((AddR (I n) ()):s, I n')) = (R (s, I (n + n')))
eval1 (E (s, Mul e1 e2)) = (E ((MulL () e2):s, e1))
eval1 (R ((MulL () e2):s, I n)) = (E ((MulR (I n)()):s, e2))
eval1 (R ((MulR (I n) ()):s, I n')) = (R (s, I (n * n')))
eval1 (E (s, And e1 e2)) = (E ((AndL () e2):s, e1))
eval1 (R ((AndL () e2):s, B b)) = (E ((AndR (B b) ()):s,
↪  e2))
eval1 (R ((AndR (B b) ()):s, B b')) = (R (s, B (b && b')))
eval1 (E (s, Or e1 e2)) = (E ((OrL () e2):s, e1))
eval1 (R ((OrL () e2):s, B b)) = (E ((OrR (B b) ()):s, e2))
eval1 (R ((OrR (B b) ()):s, B b')) = (R (s, B (b || b')))
eval1 (E (s, Lt e1 e2)) = (E ((LtL () e2):s, e1))
eval1 (R ((LtL () e2):s, I n)) = (E ((LtR (I n) ()):s, e2))
eval1 (R ((LtR (I n) ()):s, I n')) = (R (s, B (n < n')))
eval1 (E (s, Gt e1 e2)) = (E ((GtL () e2):s, e1))
eval1 (R ((GtL () e2):s, I n)) = (E ((GtR (I n) ()):s, e2))
eval1 (R ((GtR (I n) ()):s, I n')) = (R (s, B (n' < n)))
eval1 (E (s, Eq e1 e2)) = (E ((EqL () e2):s, e1))
eval1 (R ((EqL () e2):s, I n)) = (E ((EqR (I n) ()):s, e2))
eval1 (R ((EqR (I n) ()):s, I n')) = (R (s, B (n' == n)))
eval1 (E (s, If b e1 e2)) = (E ((IfF () e1 e2):s, b))
eval1 (R ((IfF () e1 _):s, (B True))) = (E (s, e1))
eval1 (R ((IfF () _ e2):s, (B False))) = (E (s, e2))
eval1 (E (s, Let x e1 e2)) = (E ((LetF x () e2):s, e1))
eval1 (R ((LetF x () e2):s, v)) = (E (s, subst e2 (x,v)))
eval1 (E (s, Error)) = (U (s, Error))--creo que falta una
↪  r...
```

```haskell
eval1 (E (s, Catch e1 e2)) = (E ((CatchL () e2):s, e1))
eval1 (R ((CatchL () _):s, I n)) = (E (s, I n))
eval1 (R ((CatchL () _):s, B b)) = (E (s, B b))
eval1 (U ((CatchL () e2):s, Error)) = (E  (s, e2))
eval1 (U (_:s, e)) = (U (s, e))
eval1 (R (_:s, _)) = (E (s, Error))

-- | evals. Recibe un estado de la maquina K y devuelve un
-- |        estado derivado de evaluar varias veces hasta
-- |        obtener la pila vacia.
evals :: State -> State
evals (E ([], I n)) = (R ([], I n))
evals (E ([], B b)) = (R ([], B b))
evals (E ([], V v)) = (R ([], V v))
evals (R ([], I n)) = (R ([], I n))
evals (R ([], B b)) = (R ([], B b))
evals (R ([], V v)) = (R ([], V v))
evals (E ([], Error)) = (E ([], Error))
evals (U ([], Error)) = (U ([], Error))
evals otra = evals(eval1 otra)

-- | eval. Recibe una expresion EAB, la evalua con la
-- |       maquina K, y devuelve un valor, iniciando con la
-- |       pila vacia esta devuelve un valor a la pila.
eval :: Expr -> Expr
eval e = let
  x = evals(E ([], e))
  in
    let ex = takeExpr x
    in
      ex

-- | vt. Funcion que verifica el tipado de un programa.
vt :: TypCtx -> Expr -> Type -> Bool
vt [] (V _) _ = False
vt ((a,b):xs) (V x) t = if x == a
                        then b == t
                        else vt xs (V x) t
vt _ (I _) t = t == Integer
vt _ (B _) t = t == Boolean
```

```
vt _ (Error) _ = True
vt s (Add e1 e2) t = t == Integer && vt s e1 t && vt s e2 t
vt s (Mul e1 e2) t = t == Integer && vt s e1 t && vt s e2 t
vt s (Succ e) t = t == Integer && vt s e t
vt s (Pred e) t = t == Integer && vt s e t
vt s (And e1 e2) t = t == Boolean && vt s e1 t && vt s e2 t
vt s (Or e1 e2) t = t == Boolean && vt s e2 t && vt s e1 t
vt s (Not e) t = t == Boolean && vt s e t
vt s (Lt e1 e2) t = t == Boolean && vt s e1 Integer && vt s
↪  e2 Integer
vt s (Gt e1 e2) t = t == Boolean && vt s e1 Integer && vt s
↪  e2 Integer
vt s (Eq e1 e2) t = t == Boolean && vt s e1 Integer && vt s
↪  e2 Integer
vt s (If b e1 e2) t = vt s b Boolean && vt s e1 t && vt s e2
↪  t
vt s (Let x e1 e2) t = vt s (subst e2 (x, e1)) t
vt s (Catch e1 e2) t = vt s e1 t && vt s e2 t
```

0.12. XEAB

En el módulo `XEAB` implementamos la siguiente extensión del lenguaje `EAB`. Ahora el argumento de la expresión *Raise* se evaluará para determinar el valor que pasará al manejador. La expresión Handle liga la variable x a la expresión manejadora (segunda expresión). El valor de la excepción estará ligado en la expresión manejadora en caso de que se genere una excepción cuando se evalúa la primera expresión. Añadimos a `data Expr` lo siguiente.

```
data Expr = V Identifier | ... | ...
          | Raise Expr
          | Handle Expr Identifier Expr
          | Write String Expr
```

Agregamos los siguientes tipos

```
type Decl = (Identifier, Type)

type TypCtxt = [Decl]
```

Agregamos lo siguiente para poder completar los tipos.

```
data Type = Integer | Boolean deriving (Eq)
```

Para representar los estados, agregamos el siguiente en `data State`.

```
data State = ... | U (State, Expr)
```

Finalmente añadimos en `data Frame` lo correspondiente a las expresiones nuevas.

```
data Frame = ... | RaiseM Pending
           | HandleM Pending String Expr
```

Las siguientes son funciones recursivas para representar este tipo de errores.

```
-- | frVars. Obtiene el conjunto de variables libres de una
-- |         expresion.
frVars :: Expr -> [Identifier]
frVars (V x) = [x]
frVars (I _) = []
frVars (B _) = []
frVars (Add a b) = frVars a `union` frVars b
frVars (Mul a b) = frVars a `union` frVars b
frVars (Succ x) = frVars x
frVars (Pred x ) = frVars x
frVars (Not x) = frVars x
frVars (And p q) = frVars p `union` frVars q
frVars (Or p q) = frVars p `union` frVars q
frVars (Lt a b) = frVars a `union` frVars b
frVars (Gt a b) = frVars a `union` frVars b
frVars (Eq a b) = frVars a `union` frVars b
frVars (If b p q) = frVars b `union` frVars p `union` frVars
↪ q
frVars (Let x p q) = frVars p `union` ([y | y <- frVars q, y
↪ /= x])
frVars (Raise a) = frVars a
```

```
frVars (Handle a x b) = frVars a `union` ([y | y <- frVars
↪  b, y /= x])
frVars (Write s e) = []

-- | subst. Realiza la substitucion de una expresion de EAB.
subst :: Expr -> Substitution -> Expr
subst (V x) (y, e) = if (x == y)
                     then e
                     else V x
subst (I n) _ = (I n)
subst (B b) _ = (B b)
subst (Add a b) s = Add(subst a s)(subst b s)
subst (Mul a b) s = Mul(subst a s)(subst b s)
subst (Succ x) s = Succ(subst x s)
subst (Pred x) s = Pred(subst x s)
subst (Not x) s = Not(subst x s)
subst (And p q) s = And(subst p s)(subst q s)
subst (Or p q) s = Or(subst p s)(subst q s)
subst (Lt a b) s = Lt(subst a s)(subst b s)
subst (Gt a b) s = Gt(subst a s)(subst b s)
subst (Eq a b) s = Eq(subst a s)(subst b s)
subst (If b p q) s = If(subst b s)(subst p s)(subst q s)
subst (Let x e1 e2) (y,e) = if(elem x ([y] ++  frVars e))
                            then error "Could not apply the
                            ↪  substitution"
                            else Let x (subst e1 (y,e))
                            ↪  (subst e2 (y,e))
subst (Raise a) s = Raise(subst a s)
subst (Handle e1 x e2) (y,e) = if(elem x ([y] ++ frVars e))
                               then error "Could not apply
                               ↪  the substitution"
                               else Handle (subst e1 (y,e))
                               ↪  x (subst e2 (y,e))
subst (Write st e) s = Write st (subst e s)

-- | eval1. Recibe un estado de la maquina K, y devuelve un
-- |         paso de la transicion.
eval1 :: State -> State
eval1 (E (s, I n)) = (R (s, I n))
eval1 (E (s, B b)) = (R (s, B b))
```

```
eval1 (E (s, V v)) = (R (s, V v))
eval1 (E (s, Succ e1)) = (E ((SuccF ()):s, e1))
eval1 (R ((SuccF ()):s, (I n))) = (R (s, I (n + 1)))
eval1 (E (s, Pred e1)) = (E ((PredF ()):s, e1))
eval1 (R ((PredF ()):s, (I n))) = (R (s, I (n - 1)))
eval1 (E (s, Not e1)) = (E ((NotF ()):s, e1))
eval1 (R ((NotF ()):s, (B b))) = (R (s, B (not b)))
eval1 (E (s, Add e1 e2)) = (E ((AddL () e2):s, e1))
eval1 (R ((AddL () e2):s, I n)) = (E ((AddR (I n) ()):s,
↪  e2))
eval1 (R ((AddR (I n) ()):s, I n')) = (R (s, I (n + n')))
eval1 (E (s, Mul e1 e2)) = (E ((MulL () e2):s, e1))
eval1 (R ((MulL () e2):s, I n)) = (E ((MulR (I n)()):s, e2))
eval1 (R ((MulR (I n) ()):s, I n')) = (R (s, I (n * n')))
eval1 (E (s, And e1 e2)) = (E ((AndL () e2):s, e1))
eval1 (R ((AndL () e2):s, B b)) = (E ((AndR (B b) ()):s,
↪  e2))
eval1 (R ((AndR (B b) ()):s, B b')) = (R (s, B (b && b')))
eval1 (E (s, Or e1 e2)) = (E ((OrL () e2):s, e1))
eval1 (R ((OrL () e2):s, B b)) = (E ((OrR (B b) ()):s, e2))
eval1 (R ((OrR (B b) ()):s, B b')) = (R (s, B (b || b')))
eval1 (E (s, Lt e1 e2)) = (E ((LtL () e2):s, e1))
eval1 (R ((LtL () e2):s, I n)) = (E ((LtR (I n) ()):s, e2))
eval1 (R ((LtR (I n) ()):s, I n')) = (R (s, B (n < n')))
eval1 (E (s, Gt e1 e2)) = (E ((GtL () e2):s, e1))
eval1 (R ((GtL () e2):s, I n)) = (E ((GtR (I n) ()):s, e2))
eval1 (R ((GtR (I n) ()):s, I n')) = (R (s, B (n' < n)))
eval1 (E (s, Eq e1 e2)) = (E ((EqL () e2):s, e1))
eval1 (R ((EqL () e2):s, I n)) = (E ((EqR (I n) ()):s, e2))
eval1 (R ((EqR (I n) ()):s, I n')) = (R (s, B (n' == n)))
eval1 (E (s, If b e1 e2)) = (E ((IfF () e1 e2):s, b))
eval1 (R ((IfF () e1 _):s, (B True))) = (E (s, e1))
eval1 (R ((IfF () _ e2):s, (B False))) = (E (s, e2))
eval1 (E (s, Let x e1 e2)) = (E ((LetF x () e2):s, e1))
eval1 (R ((LetF x () e2):s, v)) = (E (s, subst e2 (x,v)))
eval1 (E (s, Raise e)) = (E ((RaiseM ()):s, e))
eval1 (R ((RaiseM ()):s, (I n))) = (U (s,  Raise (I n)))
eval1 (R ((RaiseM ()):s, (B b))) = (U (s,  Raise (B b)))
eval1 (E (s, Handle e1 x e2)) = (E ((HandleM () x e2):s,
↪  e1))
```

```
eval1 (R ((HandleM () _ _):s, (I n))) = (R (s, (I n)))
eval1 (R ((HandleM () _ _):s, (B b))) = (R (s, (B b)))
eval1 (U ((HandleM () x e2):s, Raise(v))) = (E (s, subst e2
↪  (x,v)))
eval1 (U ((_:s), Raise(v))) = (U (s, Raise(v) ) )
eval1 (E (_, Write m e)) = (E ([],Write m e))
eval1 (R ((SuccF ()):s, (a))) = E (s, Write "Error" (Succ
↪  (a)))
eval1 (R ((PredF ()):s, (a))) = E (s, Write "Error" (Succ
↪  (a)))
eval1 (R ((NotF ()):s, (a))) = E (s, Write "Error" (Succ
↪  (a)))
eval1 (R ((AddL () e2):s, a)) = E (s, Write "Error" (Add (a)
↪  e2))
eval1 (R ((AddR (I n) ()):s, a)) = E (s, Write "Error" (Add
↪  (I n) (a)))
eval1 (R ((MulL () e2):s, a)) = E (s, Write "Error" (Mul (a)
↪  e2))
eval1 (R ((MulR (I n) ()):s, a)) = E (s, Write "Error" (Mul
↪  (I n) (a)))
eval1 (R ((AndL () e2):s, a)) = E (s, Write "Error" (And (a)
↪  e2))
eval1 (R ((AndR (B b) ()):s, a)) = E (s, Write "Error" (And
↪  (B b) (a)))
eval1 (R ((OrL () e2):s, a)) = E (s, Write "Error" (Or (a)
↪  e2))
eval1 (R ((OrR (B b) ()):s, a)) = E (s, Write "Error" (Or (B
↪  b) (a)))
eval1 (R ((LtL () e2):s, a)) = E (s, Write "Error" (Lt (a)
↪  e2))
eval1 (R ((LtR (I n) ()):s, a)) = E (s, Write "Error" (Lt (I
↪  n) (a)))
eval1 (R ((GtL () e2):s, a)) = E (s, Write "Error" (Gt (a)
↪  e2))
eval1 (R ((GtR (I n) ()):s, a)) = E (s, Write "Error" (Gt (I
↪  n) (a)))
eval1 (R ((EqL () e2):s, a)) = E (s, Write "Error" (Eq (a)
↪  e2))
eval1 (R ((EqR (I n) ()):s, a)) = E (s, Write "Error" (Eq (I
↪  n) (a)))
```

```
eval1 (R ((IfF () e1 e2):s, (a))) = E (s, Write "Error" (If
↪  (a) e1 e2))

-- | evals. Recibe un estado de la maquina K y devuelve un
-- |        estado derivado de evaluar varias veces hasta
-- |        obtener la pila vacia.
evals :: State -> State
evals (E ([], B b)) = (R ([], B b))
evals (E ([], V v)) = (R ([], V v))
evals (R ([], I n)) = (R ([], I n))
evals (R ([], B b)) = (R ([], B b))
evals (R ([], V v)) = (R ([], V v))
evals (E ([], Write m e)) = (R ([], Write m e))
evals otra = evals(eval1 otra)

-- | eval. Recibe una expresion EAB, la evalua con la
-- |       maquina K, y devuelve un valor, iniciando con la
-- |       pila vacia esta devuelve un valor a la pila.
eval :: Expr -> Expr
eval e = let
  x = evals(E ([], e))
  in
    let ex = takeExpr x
    in
      ex
```

0.13. KEAB

En el módulo `KEAB` implementamos la siguiente extensión del lenguaje `EAB`. Las continuaciones constituyen una técnica la cual proporciona una manera simple y natural de modificar el flujo de evaluación de una evaluación en lenguajes funcionales. La idea básica detrás de las continuaciones es la de considerar a la pila de control de un programa como un valor el cual puede devolverse o pasarse como argumento a otra función.

```
data Expr = V Identifier | ... | ...
          | LetCC Identifier Expr Expr
          | Continue Expr Expr
```

```
            | Cont Stack
```

Los estados se quedan igual.
Las siguientes son funciones recursivas para representar este tipo de errores.

```
-- | frVars. Obtiene el conjunto de variables libres de una
-- |        expresion.
frVars :: Expr -> [Identifier]
frVars (V x) = [x]
frVars (I _) = []
frVars (B _) = []
frVars (Add a b) = frVars a `union` frVars b
frVars (Mul a b) = frVars a `union` frVars b
frVars (Succ x) = frVars x
frVars (Pred x ) = frVars x
frVars (Not x) = frVars x
frVars (And p q) = frVars p `union` frVars q
frVars (Or p q) = frVars p `union` frVars q
frVars (Lt a b) = frVars a `union` frVars b
frVars (Gt a b) = frVars a `union` frVars b
frVars (Eq a b) = frVars a `union` frVars b
frVars (If b p q) = frVars b `union` frVars p `union` frVars
↪  q
frVars (Let x p q) = frVars p `union` ([y | y <- frVars q, y
↪  /= x])
frVars (LetCC x e) = [y | y <- frVars e, y /= x]
frVars (Continue a b) = frVars a `union` frVars b
frVars (Cont s) = []

-- | subst. Realiza la substitucion de una expresion de EAB.
subst :: Expr -> Substitution -> Expr
subst (V x) (y, e) = if (x == y)
                     then e
                     else V x
subst (I n) _ = (I n)
subst (B b) _ = (B b)
subst (Add a b) s = Add(subst a s)(subst b s)
subst (Mul a b) s = Mul(subst a s)(subst b s)
subst (Succ x) s = Succ(subst x s)
```

```
subst (Pred x) s = Pred(subst x s)
subst (Not x) s = Not(subst x s)
subst (And p q) s = And(subst p s)(subst q s)
subst (Or p q) s = Or(subst p s)(subst q s)
subst (Lt a b) s = Lt(subst a s)(subst b s)
subst (Gt a b) s = Gt(subst a s)(subst b s)
subst (Eq a b) s = Eq(subst a s)(subst b s)
subst (If b p q) s = If(subst b s)(subst p s)(subst q s)
subst (Let x e1 e2) (y,e) = if(elem x ([y] ++  frVars e))
                            then error "Could not apply the
                            ↪  substitution"
                            else Let x (subst e1 (y,e))
                            ↪  (subst e2 (y,e))
subst (LetCC x e1) (y, e) = if (elem x ([y] ++ frVars e))
                           then error "Could not apply the
                           ↪  substitution"
                           else LetCC x (subst e1 (y,e))
subst (Continue a b) s = Continue (subst a s) (subst b s)
subst (Cont _) _ = error "Could not apply the substitution"

-- | eval1. Recibe un estado de la maquina K, y devuelve un
-- |        paso de la transicion.
eval1 :: State -> State
eval1 (E (s, I n)) = (R (s, I n))
eval1 (E (s, B b)) = (R (s, B b))
eval1 (E (s, V v)) = (R (s, V v))
eval1 (E (s, Succ e1)) = (E ((SuccF ()):s, e1))
eval1 (R ((SuccF ()):s, (I n))) = (R (s, I (n + 1)))
eval1 (E (s, Pred e1)) = (E ((PredF ()):s, e1))
eval1 (R ((PredF ()):s, (I n))) = (R (s, I (n - 1)))
eval1 (E (s, Not e1)) = (E ((NotF ()):s, e1))
eval1 (R ((NotF ()):s, (B b))) = (R (s, B (not b)))
eval1 (E (s, Add e1 e2)) = (E ((AddL () e2):s, e1))
eval1 (R ((AddL () e2):s, I n)) = (E ((AddR (I n) ()):s,
↪  e2))
eval1 (R ((AddR (I n) ()):s, I n')) = (R (s, I (n + n')))
eval1 (E (s, Mul e1 e2)) = (E ((MulL () e2):s, e1))
eval1 (R ((MulL () e2):s, I n)) = (E ((MulR (I n)()):s, e2))
eval1 (R ((MulR (I n) ()):s, I n')) = (R (s, I (n * n')))
eval1 (E (s, And e1 e2)) = (E ((AndL () e2):s, e1))
```

```
eval1 (R ((AndL () e2):s, B b)) = (E ((AndR (B b) ()):s,
↪  e2))
eval1 (R ((AndR (B b) ()):s, B b')) = (R (s, B (b && b')))
eval1 (E (s, Or e1 e2)) = (E ((OrL () e2):s, e1))
eval1 (R ((OrL () e2):s, B b)) = (E ((OrR (B b) ()):s, e2))
eval1 (R ((OrR (B b) ()):s, B b')) = (R (s, B (b || b')))
eval1 (E (s, Lt e1 e2)) = (E ((LtL () e2):s, e1))
eval1 (R ((LtL () e2):s, I n)) = (E ((LtR (I n) ()):s, e2))
eval1 (R ((LtR (I n) ()):s, I n')) = (R (s, B (n < n')))
eval1 (E (s, Gt e1 e2)) = (E ((GtL () e2):s, e1))
eval1 (R ((GtL () e2):s, I n)) = (E ((GtR (I n) ()):s, e2))
eval1 (R ((GtR (I n) ()):s, I n')) = (R (s, B (n' < n)))
eval1 (E (s, Eq e1 e2)) = (E ((EqL () e2):s, e1))
eval1 (R ((EqL () e2):s, I n)) = (E ((EqR (I n) ()):s, e2))
eval1 (R ((EqR (I n) ()):s, I n')) = (R (s, B (n' == n)))
eval1 (E (s, If b e1 e2)) = (E ((IfF () e1 e2):s, b))
eval1 (R ((IfF () e1 _):s, (B True))) = (E (s, e1))
eval1 (R ((IfF () _ e2):s, (B False))) = (E (s, e2))
eval1 (E (s, Let x e1 e2)) = (E ((LetF x () e2):s, e1))
eval1 (R ((LetF x () e2):s, v)) = (E (s, subst e2 (x,v)))
eval1 (E (s, Cont r)) = (R (s, Cont r))
eval1 (E (s, LetCC x e)) = (E (s, subst e (x, Cont(s))))
eval1 (E (s, Continue e1 e2)) = (E ((ContinueL () e2):s,
↪  e1))
eval1 (R ((ContinueL () e2):s, ( n))) = (E ((ContinueR ( n)
↪  ()):s, e2))
eval1 (R ((ContinueR (Cont s') ()):s, ( n))) = (R (s', (
↪  n)))

-- | evals. Recibe un estado de la maquina K y devuelve un
-- |        estado derivado de evaluar varias veces hasta
-- |        obtener la pila vacia.
evals :: State -> State
evals (E ([], I n)) = (R ([], I n))
evals (E ([], B b)) = (R ([], B b))
evals (E ([], V v)) = (R ([], V v))
evals (R ([], I n)) = (R ([], I n))
evals (R ([], B b)) = (R ([], B b))
evals (R ([], V v)) = (R ([], V v))
evals otra = evals(eval1 otra)
```

```haskell
-- | eval. Recibe una expresion EAB, la evalua con la
-- |       maquina K, y devuelve un valor, iniciando con la
-- |       pila vacia esta devuelve un valor a la pila.
eval :: Expr -> Expr
eval e = let
  x = evals(E ([], e))
  in
    let ex = takeExpr x
    in
      ex
```

Práctica Extra 1

0.14. Índices de De-Bruijn

Dada una expresión lambda, representaremos variables ligadas apuntando directamente al símbolo lambda que la liga en el árbol de sintaxis abstracta correspondiente, es decir, mediante el número de lambdas que es necesario "saltar" hasta encontrar la lambda que liga a la variable en cuestión. Estos números, son conocidos como los índices de `De Brujin`.

```haskell
data ExprL = VL Identifier
| LL Identifier ExprL
| AL ExprL ExprL deriving (Show)

data ExprB = IB Index
| LB ExprB
| AB ExprB ExprB deriving (Show)
```

Añadimos los siguientes tipos

```haskell
type Identifier = String

type Index = Int

type Substitution = (Index, ExprB)
```

Importamos

```haskell
import Data.List
```

Las siguientes son funciones recursivas para numeros de `Brujin`.

```
-- | ctx. Funcion que obtiene el contexto canonico de una
-- |      expresion.
ctx :: ExprL -> [Identifier]
ctx (VL v) = [v]
ctx (LL i ex) = [y | y <- ctx ex, y /= i]
ctx (AL e1 e2) = ctx e1 `union` ctx e2

-- | qn. Dado un contexto de indices y una expresion lambda
-- |     obtiene su representacion anonima.
qn :: ([Identifier], ExprL) -> ExprB
qn (s, (VL i)) = IB (indice s i 0)
qn (s, (LL i e)) = LB (qn ([i]++s, e))
qn (s, (AL e1 e2)) = AB (qn (s, e1)) (qn (s, e2))

-- | newVar. Dado un contexto de nombres, obtiene una nueva
-- |         variable y la agrega al contexto.
newVar :: [Identifier] -> [Identifier]
newVar l = case l of
             [] -> ["x"]
             (x:xs) -> let otro = incrVar x
                       in
                         if otro `elem` xs
                         then [x] ++ newVar xs
                         else [x, otro] ++ xs

-- | pn. Dado un contexto de nombres y una expresion anonima
-- |     devuelve su representacion correspondiente en el
-- |     calculo lambda con nombres.
pn :: ([Identifier], ExprB) -> ExprL
pn (s, (IB i)) = VL (toma s i)
pn (s, (LB e)) = let new = newVar s
                     dif = new \\ s
                     x = toma dif 0
                 in
                   LL x (pn ([x] ++ s, e))
pn (s, (AB e1 e2)) = AL(pn (s, e1))(pn (s, e2))
```

```
-- | shift. Desplaza los indices de una expresion anonima
-- |        dado un parametro de corte.
shift :: (Int, Int, ExprB) -> ExprB
shift (d, c, (IB k)) = if k < c then (IB k) else IB (k + d)
shift (d, c, (LB t)) = LB (shift(d, c + 1, t))
shift (d, c, (AB r s)) = AB(shift(d, c, r))(shift(d,c,s))

-- | subst. Aplica la substitucion a la expresion anonima.
subst :: ExprB -> Substitution -> ExprB
subst (IB n) (j, s) = if n == j then s else (IB n)
subst (LB t) (j, s) = LB(subst t (j + 1, shift(1, 0, s)))
subst (AB t r) (j, s) = AB(subst t (j, s))(subst r (j, s))

-- | eval1. Aplica un paso de la reduccion de una expresion
-- |        anonima.
eval1 :: ExprB -> ExprB
eval1 (AB (LB t)s) = shift(-1, 0, subst t (0, shift (1, 0,
↪  s)))
eval1 (LB t) = LB (eval1 t)
eval1 (AB t1 t2) = AB (t1) (eval1 t2)
eval1 (IB n) = (IB n)

-- | locked. Determina si una expresion anonima esta
-- |         bloqueada es decir, no se pueden hacer mas
-- |         reducciones.
locked :: ExprB -> Bool
locked (IB _) = True
locked (AB (LB e1) e2) = False
locked (AB e1 e2) = locked e1 && locked e2
locked (LB e) = locked e

-- | eval. Evalua una expresion anonima hasta quedar
-- |       bloqueada.
eval :: ExprB -> ExprB
eval e = let e1 = eval1 e
         in
           if locked e1
           then e1
           else eval e1
```

Las siguientes son funciones auxiliares que se utilizaron en la resolución de ese ejercicio semanal.

```
-- | incrVar. Dado un identificador, si este no termina en
↪  número, le agrega
-- |          el sufijo '1', en caso contrario toma el valor
↪  del número y lo
-- |          incrementa en 1.
incrVar :: Identifier -> Identifier
incrVar xs = if (elem (last xs) (['a'..'z']++['A'..'Z']))
             then xs ++ show 1
             else init xs ++ show((read[last xs])+1)

-- | indice. Dada una lista, y un elemento, regresa la
↪  posición del elemento.
indice :: (Eq a) => [a] -> a -> Int -> Int
indice [] _ _ = error "No esta en la lista"
indice (x:xs) y acc = if x == y
                      then acc
                      else indice xs y (acc + 1)

-- | toma. FUnción que toma el i-esimo elemento de una
↪  lista.
toma :: [a] -> Int -> a
toma [] _ = error "No hay suficientes elementos"
toma (x:xs) 0 = x
toma (x:xs) n = toma xs (n - 1)
```

Práctica Extra 2

0.15. Semantica operacional de paso grande

Para este punto, se ha estudiado la semántica operacional de paso pequeño, la cual consiste en evaluar expresiones del lenguaje de EAB un paso a la vez. Como alternativa a esta, existe otro estilo de evaluar las expresiones que consiste en definir de forma completa cómo se evalúa una expresión hasta llegar a un valor, llamamos a esto semántica operacional de paso grande.

```
data Expr = V Identifier | I Int | B Bool
| Add Expr Expr | Mul Expr Expr | Succ Expr | Pred Expr
| Not Expr | And Expr Expr | Or Expr Expr
| Lt Expr Expr | Gt Expr Expr | Eq Expr Expr
| If Expr Expr Expr
| Let Identifier Expr Expr
| Pair Expr Expr
| Fst Expr | Snd Expr
| Lam Identifier Expr
| App Expr Expr
| Rec Identifier Expr deriving (Eq)
```

Importamos

```
import Data.List
```

Añadimos los siguiente `type`

```
type Identifier = String
```

```
type Substitution = (Identifier, Expr)

type Decl = (Identifier, Type)

type TypCtx = [Decl]
```

El siguiente `data` es para poder revisar el tipado de las expresiones.

```
data Type = Integer
          | Boolean
          | Prod Type Type
          | Func Type Type deriving (Eq)
```

Las siguientes son funciones recursivas para el tipo de dato `Expr`.

```
-- | frVars. Obtiene el conjunto de variables libres de una
-- |         expresion.
frVars :: Expr -> [Identifier]
frVars (V x) = [x]
frVars (I _) = []
frVars (B _) = []
frVars (Add a b) = frVars a `union` frVars b
frVars (Mul a b) = frVars a `union` frVars b
frVars (Succ x) = frVars x
frVars (Pred x ) = frVars x
frVars (Not x) = frVars x
frVars (And p q) = frVars p `union` frVars q
frVars (Or p q) = frVars p `union` frVars q
frVars (Lt a b) = frVars a `union` frVars b
frVars (Gt a b) = frVars a `union` frVars b
frVars (Eq a b) = frVars a `union` frVars b
frVars (If b p q) = frVars b `union` frVars p `union` frVars
↪ q
frVars (Let x p q) = frVars p `union` ([y | y <- frVars q, y
↪ /= x])
frVars (Pair a b) = frVars a `union` frVars b
frVars (Fst a) = frVars a
frVars (Snd b) = frVars b
```

```
frVars (Lam x e) = [y | y <- (frVars e) , y/=x ]
frVars (App e1 e2) = [x | x <- (frVars e1 `union` frVars
↪  e2)]
frVars (Rec i ex) = [y | y <- (frVars ex) , y/=i ]

-- | subst. Aplica una sustitucion.
subst :: Expr -> Substitution -> Expr
subst (V x) (y, e) = if (x == y)
                     then e
                     else V x
subst (I n) _ = (I n)
subst (B b) _ = (B b)
subst (Add a b) s = Add(subst a s)(subst b s)
subst (Mul a b) s = Mul(subst a s)(subst b s)
subst (Succ x) s = Succ(subst x s)
subst (Pred x) s = Pred(subst x s)
subst (Not x) s = Not(subst x s)
subst (And p q) s = And(subst p s)(subst q s)
subst (Or p q) s = Or(subst p s)(subst q s)
subst (Lt a b) s = Lt(subst a s)(subst b s)
subst (Gt a b) s = Gt(subst a s)(subst b s)
subst (Eq a b) s = Eq(subst a s)(subst b s)
subst (If b p q) s = If(subst b s)(subst p s)(subst q s)
subst (Let x e1 e2) (y,e) = if(elem x ([y] ++  frVars e))
                            then error "Could not apply the
                            ↪  substitution"
                            else Let x (subst e1 (y,e))
                            ↪  (subst e2 (y,e))
subst (Pair a b) s = Pair(subst a s)(subst b s)
subst (Fst a) s = Fst(subst a s)
subst (Snd b) s = Snd(subst b s)
subst (Lam x e) (i,s)
  | x == i = (Lam x e)
  | x `elem` (frVars s) = Lam (incrVar x) (subst e (i,s))
  | otherwise = Lam x (subst e (i,s))
subst (App e e1) (i,s) = App (subst e (i,s)) (subst e1
↪  (i,s))
subst (Rec x e) (i,s)
  | x == i = (Rec x e)
  | x `elem` (frVars s) = Rec (incrVar x) (subst e (i,s))
```

```haskell
  | otherwise = Rec x (subst e (i,s))

-- | evals. Devuelve la evaluacion de una expresion
-- |        implementando las reglas anteriores(PDF).
evals :: Expr -> Expr
evals (V _) = error "Ya no se puede evaluar."
evals (I n) = (I n)
evals (B b) = (B b)
evals e@(Pair a b) = if (canonica e)
                     then e
                     else Pair(evals a)(evals b)
evals a@(Lam x e) = if (canonica a)
                    then a
                    else (Lam x (evals e))
evals (Pred a) = let n = evals a
                 in
                   if isNat n
                   then I ((tomaNat n) - 1)
                   else error "No es un numero"
evals (Succ a) = let n = evals a
                 in
                   if isNat n
                   then I ((tomaNat n) + 1)
                   else error "No es un numero"

evals (Add a b) = let n1 = evals a
                      n2 = evals b
                  in
                    if isNat n1 && isNat n2
                    then I (tomaNat n1 + tomaNat n2)
                    else error "No es un numero"
evals (Mul a b) = let n1 = evals a
                      n2 = evals b
                  in
                    if isNat n1 && isNat n2
                    then I (tomaNat n1 * tomaNat n2)
                    else error "No es un numero"
evals (Lt a b) = let n1 = evals a
                     n2 = evals b
                  in
```

```
                    if isNat n1 && isNat n2
                    then B (tomaNat n1 < tomaNat n2)
                    else error "No es un numero"
evals (Gt a b) = let n1 = evals a
                     n2 = evals b
                  in
                    if isNat n1 && isNat n2
                    then B (tomaNat n1 > tomaNat n2)
                    else error "No es un numero"
evals (Eq a b) = let n1 = evals a
                     n2 = evals b
                  in
                    if isNat n1 && isNat n2
                    then B (tomaNat n1 == tomaNat n2)
                    else error "No es un numero"
evals (Not e) = let b = evals e
                in
                  if isBool b
                  then B (not (tomaBool b))
                  else error "No es un booleano"
evals (And a b) = let b1 = evals a
                      b2 = evals b
                  in
                    if isBool b1 && isBool b2
                    then B (tomaBool b1 && tomaBool b2)
                    else error "No es un booleano"
evals (Or a b) = let b1 = evals a
                     b2 = evals b
                 in
                   if isBool b1 && isBool b2
                   then B (tomaBool b1 || tomaBool b2)
                   else error "No es un booleano"
evals (If b e1 e2) = let con = evals b
                     in
                       if isBool con
                       then if tomaBool con
                            then evals e1
                            else evals e2
                       else error "NO es un booleano"
evals (Let x e1 e2) = let c1 = evals e1
```

```
                              c2 = evals (subst e2 (x, c1))
                          in
                            c2
evals (Fst e) = let c = evals e
                in
                  if isPair c
                  then tomaFst c
                  else error "No es un par"
evals (Snd e) = let c = evals e
                in
                  if isPair c
                  then tomaSnd c
                  else error "No es un par"
evals (App e1 e2) = let o@(Lam x e1') = evals e1
                        c2 = evals e2
                        c = evals (subst o (x, c2))
                    in
                      c
evals (Rec x e) = Rec x e

-- | vt. Funcion que verifica el tipado de un programa.
vt :: TypCtx -> Expr -> Type -> Bool
vt [] (V _) _ = False
vt ((a,b):xs) (V x) t = if x == a
                        then b == t
                        else vt xs (V x) t
vt _ (I _) t = t == Integer
vt _ (B _) t = t == Boolean
vt s (Add e1 e2) t = t == Integer &&
                     vt s e1 t &&
                     vt s e2 t
vt s (Mul e1 e2) t = t == Integer &&
                     vt s e1 t &&
                     vt s e2 t
vt s (Succ e) t = t == Integer &&
                  vt s e t
vt s (Pred e) t = t == Integer &&
                  vt s e t
vt s (And e1 e2) t = t == Boolean &&
                     vt s e1 t &&
```

```
                     vt s e2 t
vt s (Or e1 e2) t = t == Boolean &&
                    vt s e2 t &&
                    vt s e1 t
vt s (Not e) t = t == Boolean &&
                 vt s e t
vt s (Lt e1 e2) t = t == Boolean &&
                    vt s e1 Integer &&
                    vt s e2 Integer
vt s (Gt e1 e2) t = t == Boolean &&
                    vt s e1 Integer &&
                    vt s e2 Integer
vt s (Eq e1 e2) t = t == Boolean &&
                    vt s e1 Integer &&
                    vt s e2 Integer
vt s (If b e1 e2) t = vt s b Boolean &&
                      vt s e1 t &&
                      vt s e2 t
vt s (Let x e1 e2) t = vt s (subst e2 (x, e1)) t
vt s (Pair a b) (Prod t1 t2) = vt s a t1 && vt s b t2
vt s (Fst a) t = vt s a t
vt s (Snd a) t = vt s a t
vt [] (Lam x e) (Func t1 t2) = error "No hay tipo para x"
vt s@((v, bi):xs) l@(Lam x e) t@(Func t1 t2) = if (v == x)
                                                then bi == t1 &&
                                                ↪  vt s e t2
                                                else vt xs l t

-- | eval. Funcion que devuelve la evaluacion de una
-- |       expresion solo si esta bien tipada.
eval :: Expr -> Expr
eval c@(If b e1 e2) = let con = evals b
                          e1' = evals e1
                          e2' = evals e2
                          st = sameType e1' e2'
                      in
                        if (isBool con)
                        then if st then evals c else error
                         ↪  "Invalid Expresion"
                        else error "No es un booleano"
```

```
eval otro = evals otro
```

Las siguientes son funciones auxiliares que se utilizaron en la resolución de ese ejercicio semanal.

```
-- | incrVar. Dado un identificador, si este no termina en
↪  número, le agrega
-- |          el sufijo '1', en caso contrario toma el valor
↪  del número y lo
-- |          incrementa en 1.

--SE HIZO PARA HACER SUSTITUCIONES
incrVar :: Identifier -> Identifier
incrVar xs = if (elem (last xs) (['a'..'z']++['A'..'Z']))
             then xs ++ show 1
             else init xs ++ show((read[last xs])+1)

sameType :: Expr -> Expr -> Bool
sameType (I _) (I _) = True
sameType (B _) (B _) = True
sameType _ _ = False

-- | isPair. Función que nos dice si una Expr es un par.
isPair :: Expr -> Bool
isPair (Pair _ _) = True
isPair _ = False

-- | tomaFst. Función que toma la primer entrada de un par.
tomaFst :: Expr -> Expr
tomaFst (Pair a _) = a
tomaFst _ = error "No es un par desde tomaFst"

-- | tomaSnd. Función que toma la segunda entrada de un par.
tomaSnd :: Expr -> Expr
tomaSnd (Pair _ b) = b
tomaSnd _ = error "No es un par desde tomaSnd"

-- | isBool. Función que nos dice si una Expr es de tipo
↪  Bool.
```

```
isBool :: Expr -> Bool
isBool (B _) = True
isBool _ = False

-- | tomaBool. Función que toma el valor de un tipo Bool
tomaBool :: Expr -> Bool
tomaBool (B b) = b
tomaBool _ = error "No es un booleano desde tomaBool"

-- | isNat. Función que nos dice una Expr es de tipo Nat.
isNat :: Expr -> Bool
isNat (I _) = True
isNat _ = False

-- | tomaNat. Función que toma el valor de un tipo Nat
tomaNat :: Expr -> Int
tomaNat (I n) = n
tomaNat _ = error "No es un numero desde tomaNat"

-- | canonica. Función que calcula el conjunto C (formas
↪  canónicas) de una Expr.
canonica :: Expr -> Bool
canonica (I _) = True
canonica (B _) = True
canonica (Pair a b) = canonica a && canonica b
canonica a@(Lam x e) = frVars a == []
canonica _ = False
```