Paradigmas de Lenguajes de Programación

Tomás Agustín Hernández

Programación Funcional

Consiste en definir funciones y aplicarlas para procesar información.
Las funciones son verdaderamente funciones (parciales):

- Aplicar una función no tiene efectos secundarios.
- A una misma entrada le corresponde siempre la misma salida.
- Las estructuras de datos son inmutables.

Las funciones, además son datos como cualquier otro:

- Se pueden pasar como parámetros.
- Se pueden devolver como resultados.
- Pueden formar parte de estructuras de datos. Ej.: Un árbol binario que en sus nodos hay funciones.

Expresiones

Son secuencias de símbolos que sirven para representar datos, funciones, y funciones aplicadas a los datos.
Una expresión puede ser:

- Un constructor: True, False, [], (:), 0, 1, 2.
  - Type Constructor: Es un constructor que se utiliza para crear un nuevo tipo.
  - Data Constructor: Se utiliza para crear valores de ese tipo.
  - Ej.: $data\ Color\ =\ Rojo \mid Verde \mid Azul$. Azul es un **data constructor** pues nos permite crear valores del tipo Color mientras que el Type Constructor es Color.
  - Ej.: $data\ Complejo\ =\ C\ Float\ Float$. C es una función que recibe dos Float y es un constructor de Complejo.
- Una variable: longitud, ordenar, x, xs $(+)$, (*).
- La aplicación de una expresión a otra: ordenar lista, not True, $(+)$ 1.

Función Parcialmente Aplicada

Una función parcialmente aplicada es una función a las cuales se llama otra función pero no se le proporcionan todos los argumentos.
Ahora, es importante que para que esta función parcialmente aplicada tenga sentido se le manden todos los parámetros.
Es una especie de $a \to b$ Ej.:

```
add :: Int -> Int -> Int
add x y = x + y

add5 :: Int -> Int
add5 = add 5

add5 es una función que ejecuta la función parcial add pues le manda solo un parámetro y necesita dos
    .
```

Una buena pregunta entonces es ¿pero por qué es una función parcial add 5? Pues hay una función add5 que la implementa sin pasarle todos los parámetros directamente explícitamente.

```
add5 3 -> Este 3 llega como "y" a add

add5 = add 5
add = 5 + y
add = 5 + 3
add = 8
```

Otro ejemplo

```
const :: a -> b -> a
const x y = x

const (const 1) 2 -> const 1
El llamado de (const 1) es una función parcialmente aplicada porque no envía el valor de y, sin
    embargo, const (const 1) 2 no la aplica parcialmente porque le termina mandando los dos pará
    metros.
```

Aplicación de Expresiones

Es asociativa hacia la izquierda:

- $f\ x\ y \equiv\ (f\ x)\ y$
- $((((f\ a)\ b)\ c)\ d)$: Primero calcula el resultado que devuelve la expresión f enviando el valor de a. Nótese que la ídea seria que (f a) devuelva una expresión del tipo función pues luego le pasamos otro parámetro (b).

Importancia de la Aplicación de las Expresiones

¿Qué sucede si aplicamos Head Tail l? Recordemos que la asociatividad a la izquierda haría algo así Head(Tail) l pero Tail en ese momento no es nada, y si aplicamos head explota.
En este caso los paréntesis son importantes: (head (tail l))

Veamos otro ejemplo $map(\backslash x \to x\ 0)\ (map(+)\ [1,2,3])$. En este caso la asociatividad a la izquierda nos muestra algo así $map(\backslash x \to x\ 0)\ ((map(+)\ [1,2,3]))$
Esto genera $map(\backslash x \to x\ 0)\ [2,3,4]$ ahora según lo que haga la función x, hace una cosa u otra. Imaginemos que lo llamamos como map (+1) [2, 3, 4] esto daría $[3,4,5]$

Función $

Aplica una función a un valor

```
1  g :: (a -> b) -> (a -> b)
2  g x y = x y
```

Construyendo una lista paso a paso con constructores

Ej.: ¿Como construimos la lista [1, 2] utilizando constructores?

- Lo primero que necesitamos, es un constructor de listas. Para eso tenemos la expresión (:). Recordando que la aplicación de expresiones es asociativo a izquierda.
- Veamos su tipo desde GHCI $(:)\ ::\ a -> [a] -> [a]$.
- Necesitamos enviarle un valor de tipo a, una lista y como resultado, la operación (:) devuelve una lista.
- Preguntemos lo siguiente: ¿Con ((:) 1) nos basta para agregar a una lista? No, porque no estamos cumpliendo el tipado del constructor. Si quisieramos una lista con solamente el 1 si bastaría, pero acá también queremos el 2.
- Entonces, comenzamos aplicando la expresión (:) con el número 2, y ahí sí enviamos como segundo parámetro una lista vacía (que da como resultado) una lista vacía.
- (((:) 2) []) = [2]
- Por último, ((:) 1) [2] también cumple el tipo pues nos quedaría ((:) 1) [2] = [1, 2]

¿Y si quisieramos construir 1, 2, 3, 4? Recordemos la asociatividad a la izquierda, pero a la hora de querer utilizar una expresión, hay que cumplir el tipado. Hasta que ese tipado no se cumpla, Haskell tratará de resolver la expresión más profunda para ver si reduciendo cumple el tipo.

$$((:)1)\ (((:)\ 4)(((:)\ 2)(((:)\ 3)\ [])))$$

Esto lo podemos ver como $a \to (b \to (c \to (d)))$ pues para conocer a, necesitamos construir la lista de la derecha, para conocer a b necesitamos la lista de la derecha, para conocer a c necesitamos la lista de d, y d inicializa la lista vacía. Esto es importantísimo porque claramente no podríamos usar (:) 1 sin una lista. Entonces Haskell evalúa todo lo de la derecha hasta que obtenga una lista. Si no se obtuviera una lista, sería inválido.

Polimorfismo

Sucede en aquellas expresiones que tienen más de un tipo.

- $id\ ::\ a \to a$
- $(:) :: a \to [a] \to [a]$

- $fst :: (a, b) \to a$
- $cargarStringArray :: String -> [] \to [String]$. No es una expresión Polimórfica.

Nota: Es importante que si tenemos 2 parámetros de tipo a significa que los dos parámetros deben ser de ese tipo. Si tenemos 2 parámetros uno de tipo a y uno de b podría suceder que sean diferentes pero puede que sean el mismo. **Importante**: Recordar que si usamos operadores, colocar las clases que correspondan. Ej.: $a > 0$ a debe ser $(Num\ a, Ord\ a)$.
Véase **anexo** para ver ejemplos de las clases de tipos.

Polimorfismo con Clases (Type Classes)

Es posible limitar el polimorfismo a clases específicas. Es decir, si nos mandan un tipo a podemos decir que ese tipo a es genérico pero de una clase específica.
Ejemplo: $func :: Num\ a \implies a \to a \to a$
En este caso, $Num\ a$ es una Type Class.

Modelo de Cómputo (cálculo de valores)

Dada una expresión, se computa su valor usando las ecuaciones siempre y cuando estén bien tipadas.
Importante: Que una expresión se cumpla el tipado, no significa que devuelva un valor.

- $sumarUno :: a -> a$: No falla nunca (función total)
- $division :: a -> b$: Falla si b = 0. Esto nos demuestra que aunque los parámetros estén bien tipados, podemos tener indefiniciones (función parcial).

¿Cómo está dado un programa en Funcional?

Un programa funcional está dado por un conjunto de **ecuaciones orientadas**. Se les llama de esta forma pues del **lado izquierdo está lo que define y del lado derecho la definición** (o expresión que produce el valor) Una ecuación e1 = e2 se interpreta desde dos puntos de vista

- **Denotacional**: Declara que e1 y e2 tienen el mismo significado.
  - "**Denotan** lo mismo"
- **Operacional**: Computar el valor de e1 se reduce a computar el valor de e2.
  - "**Operan** de la misma forma"

¿Cómo es el lado izquierdo de una ecuación orientada?

No es una expresión arbitraria. Debe ser una función aplicada a **patrones**.
Un patrón puede ser:

- Una variable: a, b, c.
- Un comodin: _.
- Un constructor aplicado a patrones: Recordemos que un constructor sería algo que construye un tipo.
  - True, False, 1, etc.

Importante: El lado izquierdo NO debe contener variables repetidas.
Ej.: $iguales\ x\ x\ =\ True$ es una expresión mal formada. Esto pues dos valores que pueden ser diferentes, no pueden caer en la misma variable.
Ej.: $predecesor\ (n+1)\ =\ n$ también está mal formada porque estamos haciendo un cálculo del lado izquierdo, y recordemos que del lado izquierdo solo hay definiciones. Del lado derecho se hacen los cómputos o cálculo de los valores.

¿Cómo evalúamos las expresiones?

- Buscamos la subexpresión más externa que coincida con el lado izquierdo de una ecuación.
- Reemplazar la subexpresión que coincide con el lado izquierdo de la ecuación por la expresión correspondiente al lado derecho.
- Continuar evaluando la expresión resultante.

¿Cuando se detiene la evaluación de una expresión?

- Cuando el programa estalla.
  - Loop infinito.
  - Indefinición.
- Cuando la expresión es una función **parcialmente aplicada**. ¿que seria esto? ¿se refiere a utilizar mal la función parcialmente aplicada?
- La expresión es un constructor o un constructor aplicado. Ej.: True, (:) 1, [1, 2, 3].
  - Yo lo veo como algo más del tipo: una fórmula atómica o algo irreducible.

¿Cómo ayuda el Lazy Evaluation (Evaluación Perezosa) a Haskell?

Muchas veces nos ayuda a evitar tocar con un valor indefinido aunque esté ahí. Como no evalúa cosas que no necesita, si hay un indefinido por ahí y no necesita ni siquiera llegar, no lo toca.

```
1    indefinido :: Int
2    indefinido = indefinido
3    head(tail [indefinido, 1, indefinido])
```

¿Qué hace tail? Toma la cola de la lista, entonces como resultado arroja [1, indefinido] (ni siquiera evaluó el primer indefinido)
¿Qué hace head ahora entonces? [1] (ni siquiera evaluó el indefinido del final)

Importancia del orden de las Ecuaciones

El criterio más fuerte debe ir por encima del resto. Porque puede haber un caso que matchee y nunca se evalúe el siguiente caso.
Veamos un ejemplo:

```
1    esCorta (_:_:_) = False
2    esCorta _ = True
```

Considerando este orden:

- Si mando una lista que siempre tiene ≥ 3 elementos da False.
- Si mando una lista que tiene < 3 elementos es siempre True.

Cambiemos el orden y veamos qué cambia

```
1    esCorta _ = True
2    esCorta (_:_:_) = False
```

Considerando este orden:

- Si la lista tiene 0, 1, 2, 3, 4, ... n elementos siempre dará True.

En la segunda aplicación ¡nunca caemos en el caso 2! y esto es importante porque sabemos que si tiene 1 cae siempre en la primera.

Notación Infija & Notación Prefija

Infija: argumento + funcion + argumento
Prefija: funcion + argumentos
Importante: $(\leq)18 \neq (\leq 18)$ pues la opción de $(\leq)18$ sería $(18 \leq x)$ mientras que (≤ 18) sería $(x \leq 18)$. Donde claramente denota que el parámetro cuando está fuera de la notación infija es el primer argumento mientras que si lo ponemos adentro hardcodeamos su posición. Es decir $(18 \leq)$ estaríamos diciendo $(18 \leq x)$

Currificación

Una función es currificada cuando tiene la siguiente forma $ab :: Int -> Int -> Int$
Esto puede parecer que recibe 2 parámetros y devuelve uno, pero en realidad lo que hace es básicamente por cada parámetro asociar a la izquierda y hacer una función por cada parámetro.
Es decir, sería algo así $ab :: Int -> (Int -> Int)$
La función Curry **NO CAMBIA** la manera en que la función original currificada recibe los argumentos, sino que, la función curry es una especie de $puente$ para que nosotros mandemos los argumentos separados y los aplique a la función no currificada.
Importante: No tiene sentido hablar de currificación cuando las funciones tienen un solo parámetro

Las funciones currificadas son realmente importantes en la Programación Funcional porque nos permite aplicarlas de forma parcial. Es una especie mas reutilizable.

Una función no currificada tiene esta pinta $suma :: (Int, Int) -> Int -> Int$ porque acá estoy obligando a la función suma a recibir una tupla de elementos, no puedo ir mandando de a uno parcialmente.
Véase **anexo** para armar una función curry y uncurry

Funciones de Orden Superior

Son funciones que reciben como parámetro otras funciones.
Definamos la composición de funciones $(g \ . \ f)$
Recordemos que en Álgebra vimos Composición de Funciones y es algo así: sean $A : B \to C$ y $D : A \to B$

La composición gof es $A \to B$. Es decir, va desde el dominio de D hasta la imagen de A. (la salida de una función debe ser la entrada de la otra.)

En Haskell, la podemos definir así

```
ent: Entrada sal: Salida

(.) :: (b -> c) -> (a -> b) -> (a -> c)

Queremos: (g . f) x => g (f x)

Desglosemos
(.) :: (ent. de g -> sal. de g) -> (ent. de f -> sal. de f) -> (ent. de f -> sal. de g)

El resultado de la composición nos da una nueva función que tiene como entrada un valor de tipo a, y
    la salida es un valor de tipo c.

Entonces, primero se evalúa f x, lo cual produce un resultado de tipo b.

Luego, este resultado (de tipo b) se pasa a g, produciendo un resultado de tipo c.

(g . f) x: envía el parámetro x a la función f, y el resultado de f x se manda a g. Esto da como
    resultado g (f x)
```

Otra forma de definir la composición es usando Notación Lambda.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
g . f = \ x -> g (f x)
```

Recordemos que esto es algo recordable pues fog seria meter g adentro de f y para eso, la salida de g debe ser la entrada de f. Y luego se devuelve como salida el dominio de g y la salida de f. Nótese que la notación lambda es súper útil para definir funciones sin nombre, que solamente hagan algo. Esto es muy útil cuando queremos mandar una función por parámetro y que una función dada la llame a esta función.

Llamados estándar vs llamados en composición

$not(null \ x) \equiv (not \ . \ null) \ x$
Esto quiere decir que si primero componemos todas las funciones que queremos aplicarle a un valor, es lo mismo que ir aplicando de una a una las funciones al argumento.
Importante: La composición funciona **sí y solo sí** alguna de las funciones está parcialmente aplicada. Ej.: $(== 0).mod \ n \ m$ no funciona, pero $(== 0).mod \ n$ sí.

Funciones Lambda

Son funciones anónimas, es decir, no tienen nombre.
($\backslash x \to x + 1$**)**: Es una función anónima que dado un x, te devuelve una función que le aplica x+1.

Funciones Lambda Anidadas

$\backslash y \to (\backslash x \to y)$: Esto a ojo es una función constante, porque dado un valor y, se aplica la función x pero se devuelve el mismo valor y.
Los parámetros se obvían en algunos casos, este es uno. La mejor forma sería $\backslash y \to \backslash x \to y$.
Mejor notación para funciones anidadas: $\backslash y \ x \to y$

Reducción de Lambda

Preguntar $foldr(\backslash x\ rec \to fx : rec) \equiv \backslash x \to (:)(fx)$ y $\backslash x \to ex \equiv e$

Asignar nombre a una función

$a = \backslash x \to x$: al nombre **a** le asigno la función anónima $\backslash x \to x$

¿Para qué queremos funciones de orden superior? Pt 1

```
dobleL :: [Float] -> [Float]
dobleL [] = []
dobleL (x:xs) = x * 2 : dobleL xs

esParL :: [Int] -> [Bool]
esParL [] = []
esParL (x:xs) = x 'mod' 2 == 0 : esParL xs
```

¿Qué es lo que tienen en común? Todas tienen una estructura bastante similar.

```
g [] = []
g (x : xs) = f x : g xs
```

Lo único que cambia es **qué se hace** en cada paso recursivo.
Hagamos una pregunta ¿la cantidad de elementos de la entrada es igual a la de la salida? en este caso sí pero ¿qué operación se está haciendo? se están haciendo ciertas manipulaciones de los datos y se devuelve la información modificada en una lista nueva. Esto, en varios lenguajes se conoce como **map**. Se hace una manipulación de los datos pero se devuelve **la misma cantidad de elementos**.

Map

Recibe como parámetro una función que es aplicada a todos los elementos y devuelve una lista nueva. Se utiliza para modificar los valores de una lista dada según lo que haga la función.

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x : xs) = f x : map f xs
```

Entonces, podemos definir **qué operación de modificación** se le realizan a los elementos de una lista dada.

```
multiplicarPorDos :: [a] -> [b]
multiplicarPorDos xs = map (\x -> x * 2) xs

dividirPorDos :: [a] -> [b]
dividirPorDos xs :: map(\x -> x / 2) xs
```

Entonces gracias a Map nos abstraemos de tener miles de funciones con la misma estructura pero solo cambien **qué hacen** con los elementos.
Véase **anexo** para ver ejercicios interesantes con Map.

¿Para qué queremos funciones de orden superior? Pt 2

```
negativos :: [Int] -> [Int]
negativos [] = []
negativos (x:xs) = if x < 0
                   then x : negativos xs
                   else negativos xs

pares :: [Int] -> [Int]
pares [] = []
pares (x:xs) = if x 'mod' 2 == 0
               then x : pares xs
               else pares xs
```

¿Qué es lo que tienen en común? Todas tienen una estructura bastante similar, pero lo único que cambia es **cuando vamos a agregar a la lista los elementos**.

```
g [] = []
g (x:xs) = f x : g xs
```

Hagamos una pregunta ¿la cantidad de elementos de la entrada es igual a la de la salida? Puede que sí, puede que no.
¿Qué operación se está haciendo? se están filtrando ciertos elementos de una lista que no cumplan una condición dada.
Esto, en varios lenguajes se conoce como **filter**. Se devuelve una nueva lista con los elementos que cumplan una condición dada.

Filter

Recibe como parámetro una función que es aplicada a todos los elementos. Se utiliza para "borrar.elementos de una lista, o quitar aquellos que no cumplan un criterio dado.

```
filter :: (a -> Bool) -> [a] -> [b]
filter p [] = []
filter p (x : xs) = if p x
                    then x : filter p xs
                    else filter p xs
```

La función que le envíamos es para especificar **qué elementos nos queremos quedar**.
Entonces, podemos definir **qué operación de filtro** se le realizan a los elementos de una lista dada.

```
eliminarImpares :: [a] -> [b]
eliminarImpares xs = filter (\x -> x `mod` 2 == 0) xs

borrarNegativos :: [a] -> [b]
borrarNegativos xs = filter(\x -> x > 0) xs
```

Entonces gracias a Filter nos abstraemos de tener miles de funciones con la misma estructura pero solo cambien **con qué elementos nos quedamos** dada una condición

Recursión

¿Qué tienen en común los siguientes problemas? ¿En qué difieren?

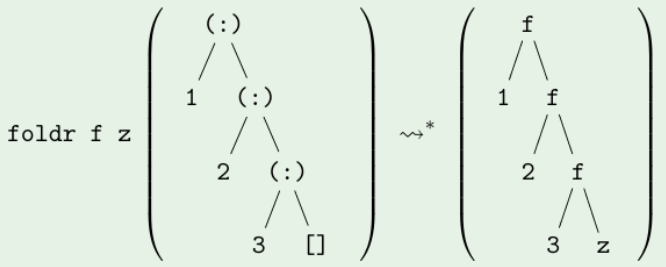
```
concat :: [[a]] -> [a]
concat [] = []
concat (x:xs) = x ++ concat xs

reverso :: [a] -> [a]
reverso [] = []
reverso (x:xs) = reverso xs ++ [x]

sum :: [int] -> int
sum [] = 0
sum (x:xs) = x + sum xs

En común:
    1. Todos tienen un caso base, pero es diferente.
    2. Todos hacen un paso recursivo, pero cambia la operación que hacen.
    3. El tipo de entrada puede no coincidir con el de la salida.
```

En este tipo de problemas lo mejor que podemos hacer es realizar una especie de función que nos permita modularizar lo más posible y aquello que difiere, pasarlo por parámetros.
Este tipo de problemas lo podemos solucionar con Foldr.

Recursión Estructural (foldr)

Se la conoce como Plegado a la Derecha porque va anidando el llamado de la función hacia adentro hasta que llega al caso base. Cuando llega al caso base, resuelve de derecha a izquierda. Una función f está dada por recursión estructural sí:

- El caso base devuelve un valor fijo z.
- El caso recursivo es una función de x y (g xs)




```
suma :: [Int] -> Int
suma = foldr (+) 0

 suma [1, 2]  ⇝   foldr (+) 0 [1, 2]
              ⇝   (+) 1 (foldr (+) 0 [2])
              ⇝   (+) 1 ((+) 2 (foldr (+) 0 []))
              ⇝   (+) 1 ((+) 2 0)
              ⇝*  3
```

- Se trabaja solamente con la cabeza de la lista.
- Se hace recursión sobre la cola pero no se tiene acceso a ella, sino al llamado recursivo. Es decir, el caso recursivo no usa xs.
- La recursión es la clásica, va desde derecha a izquierda. La R de foldr es de Right.

Su estructura formal es la siguiente

```
1  g [] = <caso base> -> valor
2  g (x:xs) = <caso recursivo> -> función cabeza de la lista y resultado de la recursión.
```

Por lo tanto foldr se define como

```
1  foldr :: (a -> (b -> b)) -> b -> [a] -> b
2  foldr f z [] = <
3  foldr f z (x:xs) = f x (foldr f z xs)
```

Importantísimo: El tipado de foldr es $Foldable\ t \implies (a \to b \to b) \to b \to ta \to b$

- a = Es el elemento que tenemos actualmente en la recursión.
- b = Es cualquier tipo que nos permita acumular. Puede ser una tupla, un número, lo que sea. Pero en cada paso recursivo hay que llenar eso. El caso base tiene que cumplir este tipo b.
- Véase **anexo** para ver ejemplos usando foldr.

¿Qué es lo que produce foldr (:) []? Es una identidad sobre listas porque haría recursión sobre una lista vacía.
¿Qué es lo que hace la siguiente función foldr? $foldr(\backslash x\ r \to x)\ 0\ [1,2]$

- Toma la lista y hace recursión. Empieza con el 2, ejecuta la función y devuelve 2. Hace el paso recursivo.
- Toma la lista y hace recursión. Ahora sigue con el 1, recibe 1 y la salida es 1.
- Esta función agarra el primer elemento de una lista, sería el head.

IMPORTANTÍSIMO: Se pueden recorrer dos listas, tres listas, las que quieras a la vez con Foldr. La recursion se hace sobre una sola, pero las n-1 listas las enviarías por argumento en cada paso recursivo. Véase **anexo** para ver ver el árbol de recursión y un ejemplo práctico. **Importante**: Foldr puede trabajar con listas infinitas.
Véase **anexo** para ver ejemplos de Foldr.

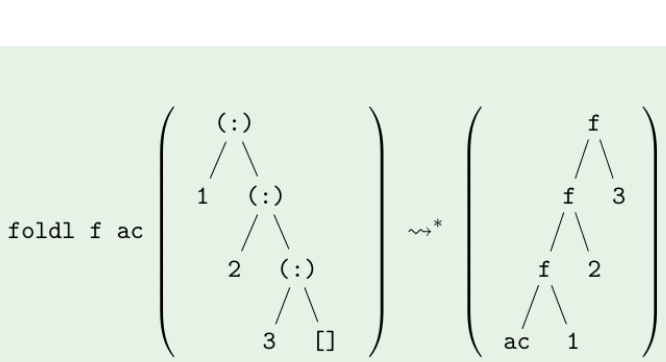
Recursión que no es estructural

```
1  ssort :: Ord a => [a] -> [a]
2  ssort [] = []
3  ssort (x:xs) = minimo (x:xs) : ssort(sacarMinimo(x:xs))
```

No es estructural pues se esta usando xs para el llamado de minimo(x:xs) y no solamente en el llamado recursivo de ssort.

Iteración (foldl)

Se la conoce como Plegado a la Izquierda porque va resolviendo inmediatamente de izquierda a derecha hasta que termine el proceso.
Este tipo de recursión es más que recursión una iteración, en este caso empezamos yendo desde el primer valor hasta el último. En este enfoque voy modificando una solución parcial.
Por lo tanto foldl se define como




```
bin2dec :: [Int] -> Int
bin2dec = foldl (\ ac b -> b + 2 * ac) 0
```

```
   bin2dec [1, 0, 0]
⇝  foldl (\ ac b -> b + 2 * ac)  0                          [1, 0, 0]
⇝  foldl (\ ac b -> b + 2 * ac)  (1 + 0)                    [0, 0]
⇝  foldl (\ ac b -> b + 2 * ac)  (0 + 2 * (1 + 0))          [0]
⇝  foldl (\ ac b -> b + 2 * ac)  (0 + 2 * (0 + 2 * (1 + 0))) []
⇝  0 + 2 * (0 + 2 * (1 + 0))
⇝* 4
```

```
1  foldl :: (b -> a -> b) -> b -> [a] -> b
2  foldl f ac [] = ac
3  foldl f ac (x:xs) = foldl f (f ac x) xs
```

¿Qué es lo que sucede en el siguiente ejemplo?

```
1  foldl (\x y -> 1) 0 unos
2  foldl f (f 0 1) unos
3  foldl f(f(f 0 1) 1) unos
```

Esto es in-realizable con foldl. Es decir, foldl no puede manejar listas infinitas.
Importantísimo: El tipado de foldl es $Foldable\ t \implies (b \to a \to b) \to b \to ta \to b$

- b = Es cualquier tipo que nos permita acumular. Puede ser una tupla, un número, lo que sea. Pero en cada paso recursivo hay que llenar eso. El caso base tiene que cumplir este tipo b.
- a = Es el elemento que tenemos actualmente en la recursión.

Véase **anexo** para ver ejemplos usando foldl.

¿Por qué foldl es peor que foldr?

foldl: Procesa la lista de izquierda a derecha, lo que requiere evaluar toda la lista antes de devolver un resultado, lo que no es posible con listas infinitas.
foldr: Procesa la lista de derecha a izquierda, permite trabajar de manera perezosa y puede manejar listas infinitas si la función y el valor inicial permiten una evaluación parcial o completa sin necesidad de procesar todos los elementos.

Recursión Primitiva (recr)

No existe en Haskell. Es una manera de nosotros podemos tener las mismas ventajas de foldr pero en este caso, la recursión primitiva nos permite utilizar la cola de la lista

- Se trabaja solamente con la cabeza de la lista y la cola.
- Se hace recursión sobre la cola.

Su estructura forma les la siguiente

```
1  g [] = b
2  g (x:xs) = f x xs (g xs)
```

Por lo tanto recr se define como

```
1  recr :: (a -> [a] -> b ->b) -> b -> [a] -> b
2  recr f z [] = z
3  recr f z (x:xs) = f x xs (recr f z xs)
```

Tipos

Existen diferentes maneras de definir tipos. Esto es según sea el objetivo. Definimos tipos con la palabra **data** + Nombre = Tipo1 — Tipo2 — Tipo 3
Si hacemos en GHCI : $tTipo1$ saldrá que es de tipo Nombre.
Nota: | indica que a continuación hay otro tipo.

Tipos Comunes

Son no recursivos, ej: $dataDia = Lu|Ma|Mi|Mie|Ju|Vi|Sa|Do$
Lu, Ma, Mi, ... son constructores del tipo dia.
Los argumentos podrán ser recibidos diciendo qué tipo se espera, y usamos pattern matching para utilizarlos.

```
EsLunes :: Dia -> Bool
EsLunes Lu = True
EsLunes _ = False

EsFinDeSemana :: Dia -> Bool
EsFinDeSemana Sa = True
EsFinDeSemana Do = True
EsFinDeSemana _ = False
```

Tipos con Funciones

Los tipos también pueden tener tipos que necesiten argumentos. Difieren en la info que devuelven.
Ej.: data Persona = LaPersona String String Int
Importante

- Si evaluamos : $tLaPersona$ sin enviar los argumentos retornará $LaPersona :: String -> String -> Int$.
- Si evaluamos : $tLaPersona"T","H",23$ enviando los argumentos retornará que LaPersona es de tipo Persona

```
Edad :: Persona -> Int
Edad (LaPersona n a e) = e

Cumpleaños :: Persona -> Persona
Cumpleaños (LaPersona n a e) = LaPersona n a (e+1)
```

Importante: Nótese que estamos devolviendo **una nueva persona**. En programación funcional no existe el concepto de "modificar.algo, sino crear algo nuevo con lo anterior y cambiarle algo.

Tipos Recursivos

Cuando tengo tipos recursivos, las funciones que los usen deben manejar casos bases y la recursión
Ej.: data Nat = Zero — Succ Nat

Tipos Polimórficos

Al igual que las funciones, podemos definir que los tipos tengan constructores de un tipo específico.

```
data List a = Vacia | Const a (List a)
```

Importante en Tipos

No se puede repetir un mismo constructor para un mismo tipo.
Es decir

```
data List = Vacia | Cons Int ListI
data List a = Vacia | Cost a (List a)

Error. No puede estar Vacia como constructor de dos tipos diferentes.
```

Listas

- Por extensión: Es dar la lista implícita escribiendo todos sus elementos. Ej. [1, 2, 3]
- Secuencias: Progresiones aritméticas en un rango particular. Ej.: [3..7] es la lista que tiene los números del 3 al 7.
- Por compresión: Se definen de la siguiente manera [expresion — selectores, condiciones]. Ej.: [(x,y) — x ¡- [0..5], y ¡- [0..3], x+y==4] es la lista de pares que tienen elementos de x e y que dan menos que 4

Estructuras Recursivas sobre Otras Estructuras de Datos

¿Cómo podemos plantear la estructura de un foldr para un árbol binario o cualquier estructura plegable que no sea una lista? Lo primero que podemos intuir es que en las listas, solo hay una recursión, sobre la lista propiamente pero en un Árbol Binario tenemos dos caminos: rama izquierda y rama derecha. Esto quiere decirque de alguna manera, tenemos que capturar 2 recursiones.

- Planteamos varios problemas acerca de esa estructura de datos y vemos qué patrones hay en común.
- Las cosas que sean diferentes, las pasamos por parámetros.

Por ejemplo, sean estas operaciones de un árbol binario

```
1  nodos :: AB -> Int
2  nodos Nil = 0
3  nodos (Bin i r d) = nodos i + 1 + nodos d
4
5  preorder :: AB -> [a]
6  preorder Nil = []
7  preorder (Bin i r d) = [r] ++ preorder i ++ preorder d
```

Podemos observar que cambia lo siguiente

- Tipos de salida: En el primer ejemplo devolvemos un int, en el segundo una lista de a. La lista de a sería el tipo que tenga el AB.
- Caso base: En uno es [] y en otro 0. Por lo tanto tenemos que admitir un tipo b para el caso base. El caso base es del mismo tipo que el tipo de salida **siempre**.
- Función que realiza: En uno hace sumas, en el otro hace concatenaciones.

¿Cómo podríamos escribir las funciones anteriormente mencionadas de manera anónima?

```
1  nodos: (\ri r rd -> ri + 1 + rd)(nodos i) r (nodos d)
2  preorder: (\ri r rd -> [r] ++ ri ++ rd) (preorder i) r (preorder d)
```

Recordemos la firma de foldr: $Foldable\ t \implies (a \to b \to b) \to b \to t\ a \to b$
Donde b era el caso base / recursivo y a el elemento actual, acá tenemos dos.
Recordemos la estructura del tipo AB: AB a = Nil — Bin (AB a) a (AB a)
Entonces nuestra función foldAB va a tener que estar preparado para recibir ambos (AB a) y a. foldAB :: (b → a → b → b) → b → AB a → b donde :t Bin: (AB a) → a → (AB) a → AB a donde

- $(b \to a \to b \to b)$
  - la primera b: recursión rama izquierda
  - la a: la raíz
  - la segunda b: recursión rama derecha
  - la tercera b: el resultado del proceso.
- b: el resultado.
- AB a: la entrada
- b: el resultado.

Es súper importante entender que la firma de la función que acepta foldAB está prácticamente atada al tipo de la estructura de dato.

Validación y Verificación de Programas

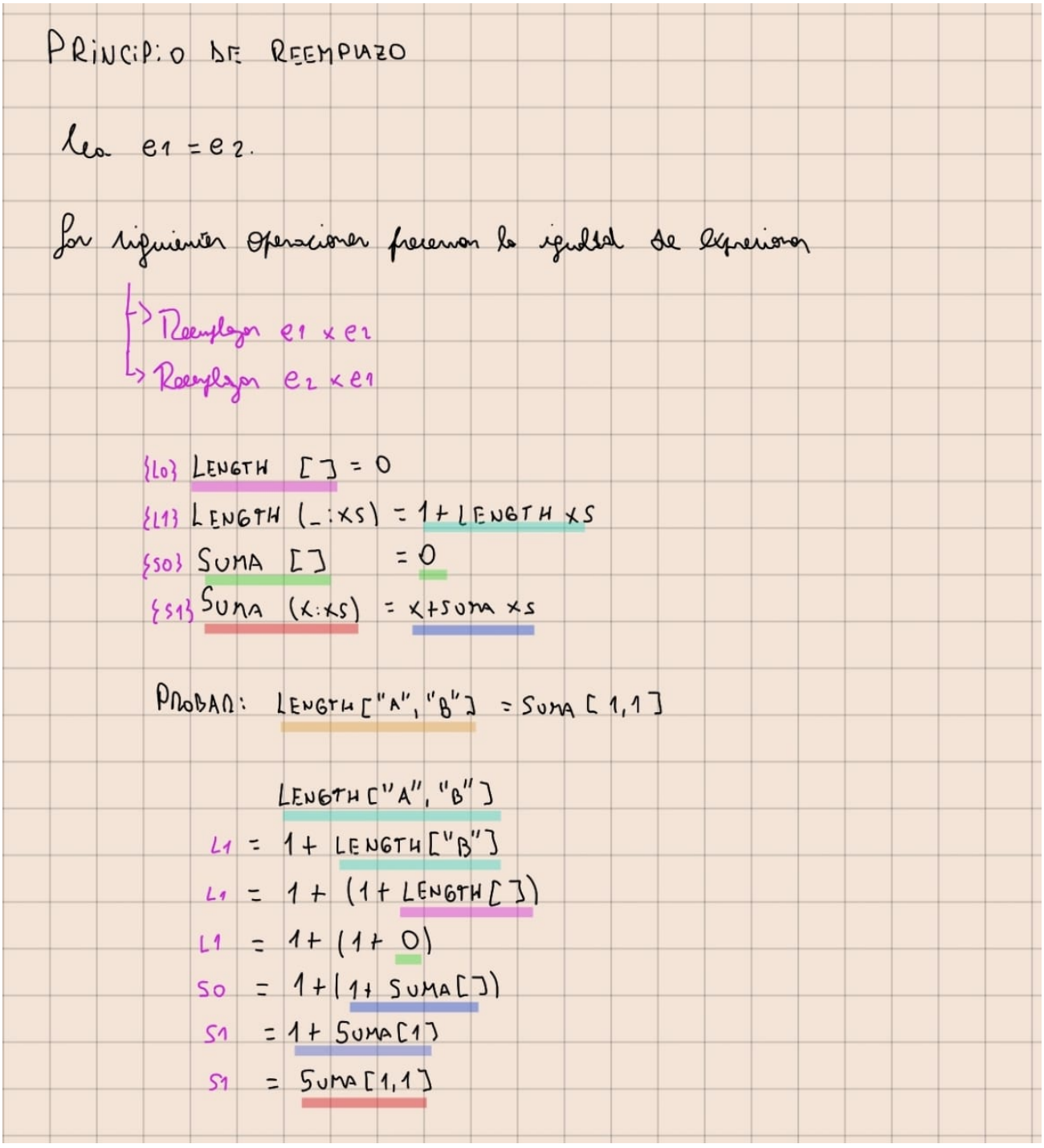
- Trabajaremos con estructuras de datos **finitas**. Más técnicamente, con tipos de datos **inductivos**.
- Trabajamos con **funciones totales**
  - Las ecuaciones deben cubrir todos los casos posibles.
  - La recursión siempre termina.
- El programa **no depende del orden** de las ecuaciones.

Principio de Reemplazo

Sea $e1 = e2$ una ecuación del programa. Las siguientes operaciones preservan la igualdad de expresiones.

- Reemplazar **cualquier instancia** de e1 por e2.
- Reemplazar **cualquier instancia** de e2 por e1.

Importante: Si una igualdad se puede demostrar usando sólo el principio de reemplazo, decimos que la igualdad vale **por definición**.



Inducción Estructural

Cada tipo de datos tiene su propio principio de inducción.
Importante: El . acá cumple el rol del () en el $\forall$ es decir $\forall x :: Bool.\mathcal{P} \equiv (\forall x :: Bool)(\mathcal{P})$ y el :: cumple el rol de **es del tipo...**
Los constructores de tipo NO recursivos serán los casos base mientras que aquellos constructores recursivos serán las hipótesis inductivas.

Formas de demostrar

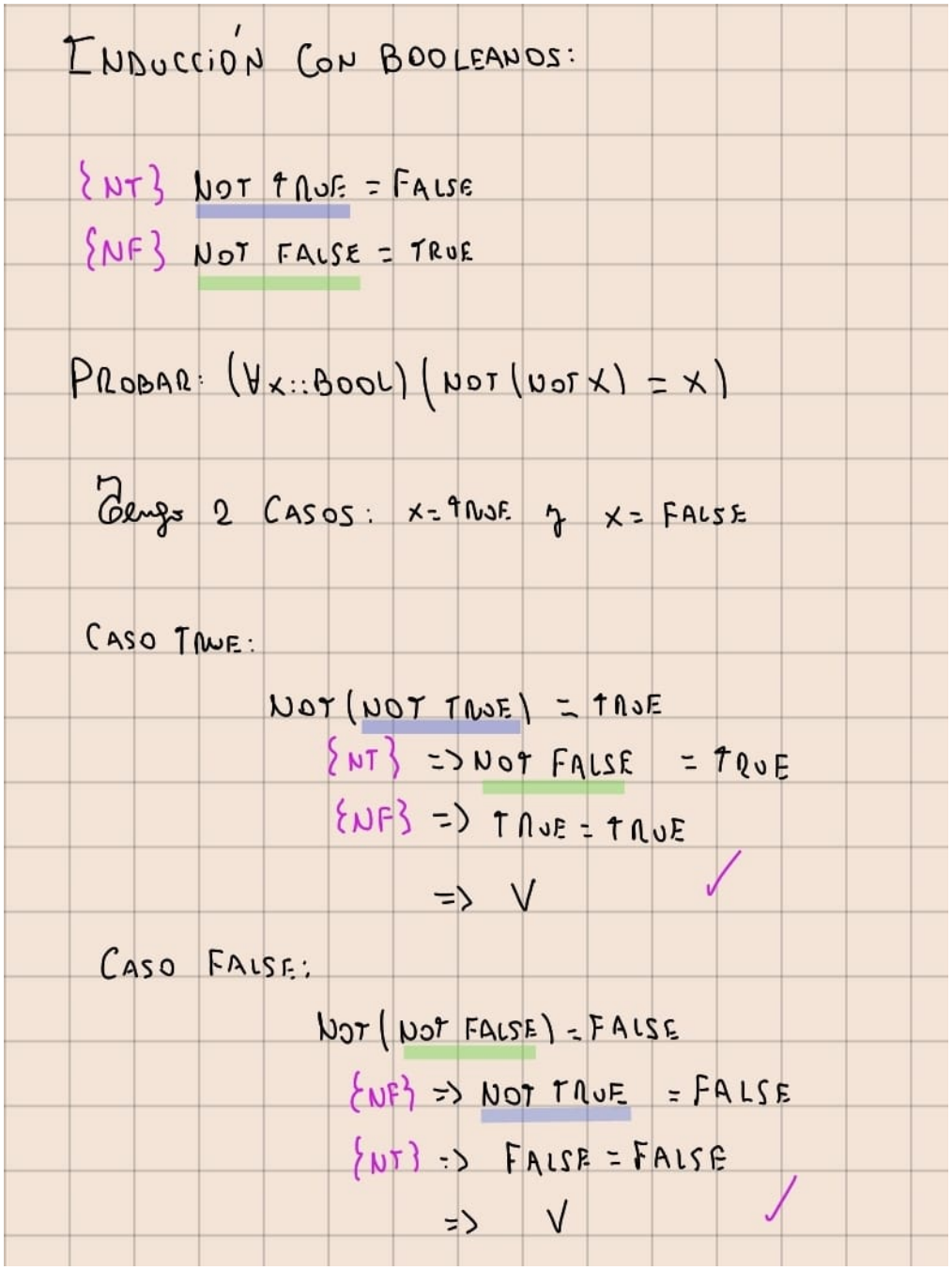
Existen 3 formas conocidas de demostrar cosas, por lo menos en la materia lo hacemos de maneras determinadas. Cuando usar una o la otra depende de la experiencia, y no es tan intuitivo saber cuando usar una opción u otra.

- Forma Álgebra: Arrancás desarrollando el lado izquierdo de tu TI. Aplicás la HI y tratás de llegar al lado derecho del TI.
- Forma Intermedia: Llegar de ambos lados de la TI a aplicar la HI. Solo funciona si estamos probando **igualdades**.
- Desarrollando lado derecho en base a contexto: Esta suele ser la opción más coherente en muchos casos. Consiste en desarrollar el TI del lado izquierdo, aplicar la HI y cuando llegamos a algo trivial o que no podemos seguir reduciendo, empezamos a desarrollar el lado derecho de la TI pero asumiendo las hipótesis del lado izquierdo.
    - Digamos que del lado izquierdo para aplicar la HI tuvimos que entrar a varios casos, por ejemplo: xs = (z:zs), e == z. Si llegamos a una respuesta trivial del lado izquierdo entonces desarrollamos el lado derecho asumiendo que xs tiene la pinta (z:zs) y efectivamente e == z. Esto nos hará todo mucho más corto.

Estas estrategias, las dejaré en el anexo para que claramente se note su diferencia y ventajas en cada caso.

Inducción sobre booleanos

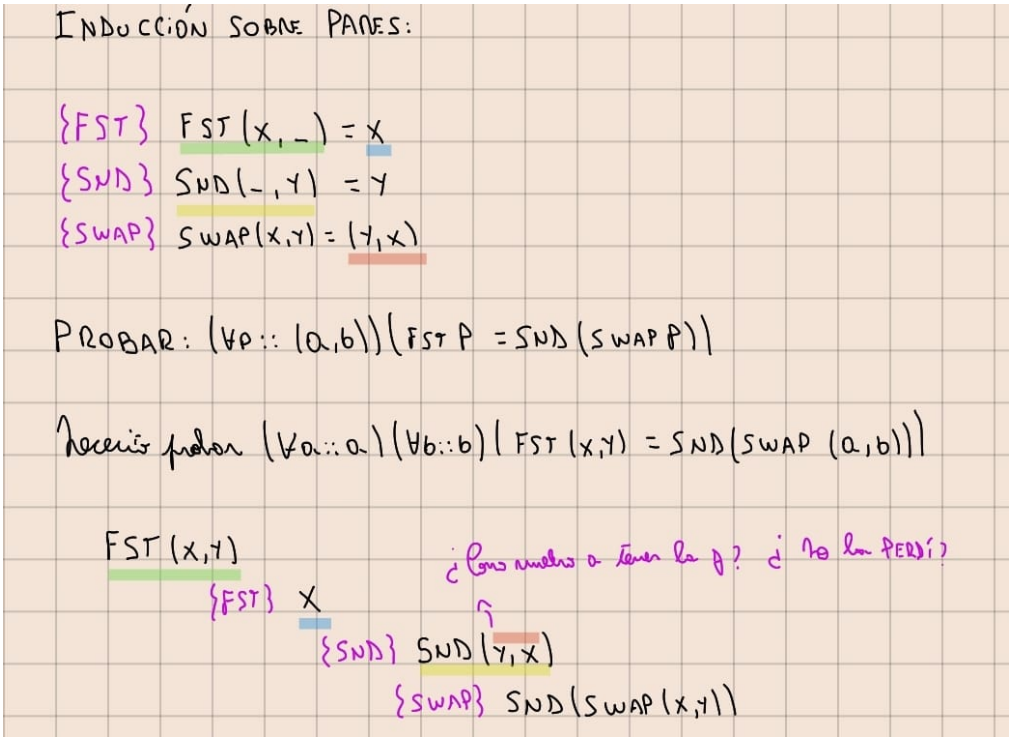
Si $\mathcal{P}(True)$ y $\mathcal{P}(False)$ entonces $\forall x :: Bool\ \mathcal{P}(x)$



Inducción sobre pares

Si $(\forall x :: a)(\forall y :: b)(\mathcal{P}(x, y))$ entonces $\forall p :: (a, b)\mathcal{P}(p)$

```
{FST}  fst (x, _)  = x
{SND}  snd (_, y)  = y
{SWAP} swap (x, y) = (y, x)
Para probar ∀p :: (a, b). fst p = snd (swap p)
basta probar:
∀x :: a. ∀y :: b. fst (x, y) = snd (swap (x, y))

fst (x, y) = x = snd (y, x) = snd (swap (x, y))
             ↑   ↑           ↑
            FST  SND        SWAP
```



Inducción sobre naturales

Si $\mathcal{P}(Zero)$ y $(\forall n :: Nat)(\mathcal{P}(n) \implies \mathcal{P}(Suc\ n))$ entonces $(\forall n :: Nat)(\mathcal{P}(n))$ donde

- $\mathcal{P}(n)$: Hipótesis Inductiva.
- $\mathcal{P}(Suc\ n)$: Tesis Inductiva.

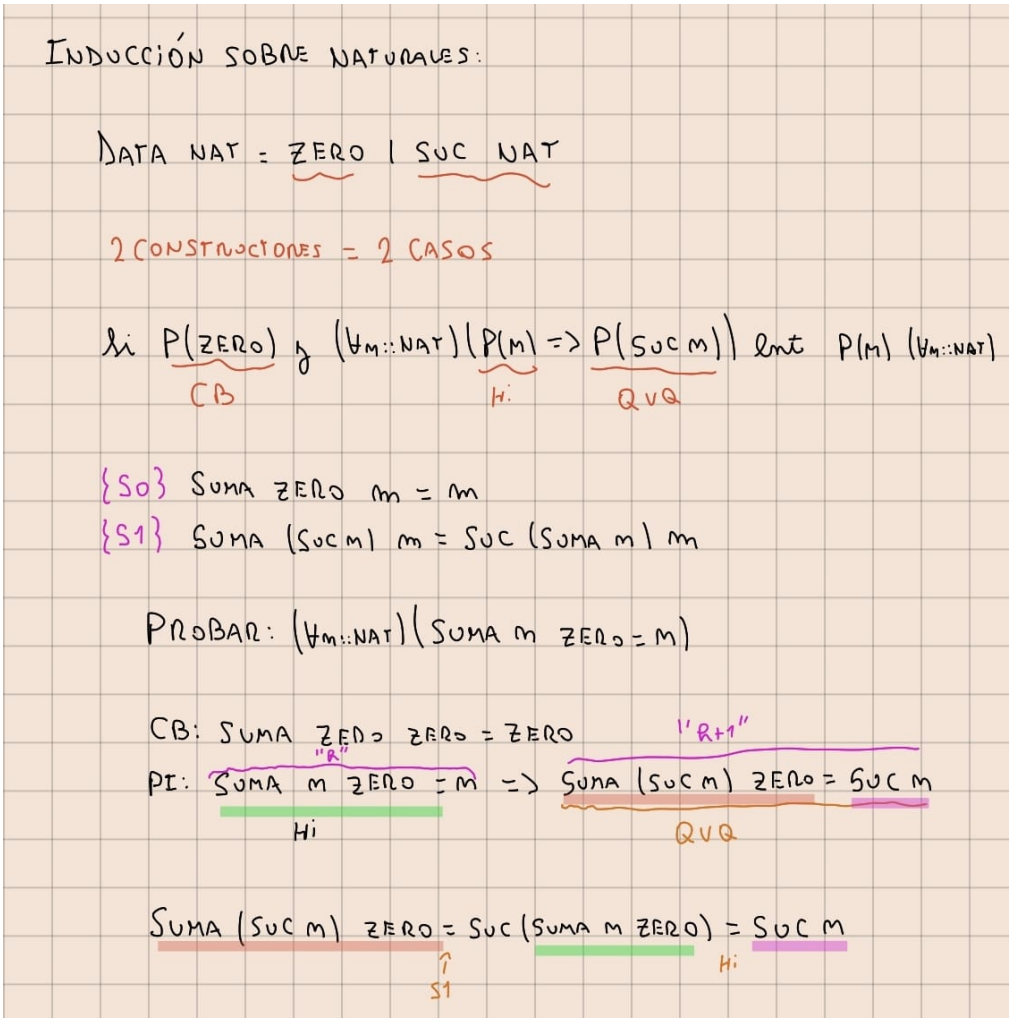
Ejemplo

```
{S0} suma Zero     m = m
{S1} suma (Suc n) m = Suc (suma n) m
```

Para probar $\forall$n :: Nat. suma n Zero = n
basta probar:

1. suma Zero Zero = Zero.
   Inmediato por S0.
2. suma n Zero = n $\Rightarrow$ suma (Suc n) Zero = Suc n.
   (suma n Zero = n: H.I. — suma (Suc n) Zero = Suc n: T.I.)

```
suma (Suc n) Zero = Suc (suma n Zero) = Suc n
                    ↑                   ↑
                    S1                  H.I.
```



Inducción Estructural: Caso General

Sea $\mathcal{P}$ una propiedad acerca de las expresiones tipo T tal que

- $\mathcal{P}$ vale sobre todos los constructores base de T,
- $\mathcal{P}$ vale sobre todos los constructores recursivos de T, asumiendo como hipótesis inductiva que vale para los parámetros de tipo T,

entonces $(\forall x :: T)(\mathcal{P}(x))$

Principio de Inducción sobre Listas

INDUCCIÓN SOBRE LISTAS:

DATA $[a] = [] \mid a : [a]$ 2 CONSTRUCTORES = 2 CASOS

si $P([])$ y $(\forall x :: a)(\forall xs :: [a])(P(xs) \Rightarrow P(x:xs))$ ent $(\forall xs :: [a])(P(xs))$

CB HI QVQ

{M0} MAP f [] = []
{M1} MAP f (x:xs) = f x : MAP f xs
{A0} [] ++ ys = ys
{A1} (x:xs) ++ ys = x : (xs ++ ys)

CB: P([])
PI: $(\forall x :: a)(\forall xs :: [a])(P(xs) \Rightarrow P(x:xs))$

PROPIEDAD: si f :: a -> b, xs :: [a], ys :: [a] ent

P = MAP f (xs ++ ys) = MAP f xs ++ MAP f ys
Hi = P(xs): MAP f (xs++ys) = MAP f xs ++ MAP f ys
QVQ = P(x:xs): MAP f ((x:xs)++ys) = MAP f (x:xs) ++ MAP f ys

P([]) = xs = []

CB: MAP f ([] ++ ys)
{A0} MAP f ys ~> (FORMA: [] ++ ys DONDE ys = MAP f ys)
{A0} [] ++ MAP f ys
{M0} MAP f [] ++ MAP f ys
se cumple P([]) => MAP f xs ++ MAP f ys pues xs = []

PI: MAP f ((x:xs) ++ ys)

{A1} MAP f (x : (xs ++ ys))
{M1} f x : MAP f (xs++ys) ys
{HI} f x : MAP f xs ++ MAP f ys (FORMA: x : (xs ++ ys))
x xs
{A1} (f x : MAP f xs) ++ MAP f ys
{M1} MAP f (x:xs) ++ MAP f ys
QVQ

Principio de Inducción sobre Árboles Binarios

Ejemplo: principio de inducción sobre árboles binarios

```
data AB a = Nil | Bin (AB a) a (AB a)
```

Sea $\mathcal{P}$ una propiedad sobre expresiones de tipo `AB a` tal que:

- $\mathcal{P}(\texttt{Nil})$
- $\forall \texttt{i} :: \texttt{AB a}.\ \forall \texttt{r} :: \texttt{a}.\ \forall \texttt{d} :: \texttt{AB a}.$
$$\underbrace{(\mathcal{P}(\texttt{i}) \wedge \mathcal{P}(\texttt{d}))}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Bin i r d})}_{\text{T.I.}}$$

Entonces $\forall \texttt{x} :: \texttt{AB a}.\ \mathcal{P}(\texttt{x})$.

Principio de Inducción sobre Polinomios

Ejemplo: principio de inducción sobre polinomios

```
data Poli a = X
            | Cte a
            | Suma (Poli a) (Poli a)
            | Prod (Poli a) (Poli a)
```

Sea $\mathcal{P}$ una propiedad sobre expresiones de tipo `Poli a` tal que:

- $\mathcal{P}(\texttt{X})$
- $\forall \texttt{k} :: \texttt{a}.\ \mathcal{P}(\texttt{Cte k})$
- $\forall \texttt{p} :: \texttt{Poli a}.\ \forall \texttt{q} :: \texttt{Poli a}.$
$$\underbrace{(\mathcal{P}(\texttt{p}) \wedge \mathcal{P}(\texttt{q}))}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Suma p q})}_{\text{T.I.}}$$
- $\forall \texttt{p} :: \texttt{Poli a}.\ \forall \texttt{q} :: \texttt{Poli a}.$
$$\underbrace{(\mathcal{P}(\texttt{p}) \wedge \mathcal{P}(\texttt{q}))}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Prod p q})}_{\text{T.I.}}$$

Entonces $\forall \texttt{x} :: \texttt{Poli a}.\ \mathcal{P}(\texttt{x})$.

Relación entre foldr y foldl

Propiedad: Si $f :: a \to b \to b,\ z :: b,\ xs :: [a]$ entonces foldr f z xs = foldl (flip f) z (reverse xs)
Lema: Si $g :: b \to a \to b,\ z :: b, x :: a, xs :: [a]$ entonces foldl g z (xs ++ [x]) = g (foldl g z xs) xs

Puntos de vista intensional vs extensional

Sí, es con **s**. Es intensional.
¿Es equivalente mergesort = insertionSort? La realidad es que: hacen lo mismo (llegan al mismo resultado); ordenan algo pero de una manera diferente.

- Punto de vista intensional: Dos valores son iguales si están definidos de la misma manera.
- Punto de vista extensional: Dos valores son iguales si son indistinguibles al observarlos.

Entonces mergesort = insertionSort desde el lado extensional, porque hacen lo mismo pero no son iguales desde el punto de vista intensional.

Principio de Extensionalidad Funcional

Sean $f,\ g :: a \to b$.
Principio de extensionalidad funcional: Si $(\forall x :: a)(f\ x = g\ x)$ entonces $f = g$
El ppio. de extensionalidad funcional nos dice que si son funciones son iguales entonces son iguales punto a punto donde las evaluemos.

Principio de Extensionalidad

Nos permite predicar sobre la forma de algo particular y nos da la ventaja de poder seguir estudiando algo particular y poder sacar ciertas conclusiones.
Importante: Es súper común tener que aplicar este tipo de estrategias durante una demostración varias veces.

Principio de Extensionalidad de Booleanos

Dada una condición booleana, difurcamos en dos casos: o es verdadero, o es falso.
Ej.: $e == x$ se separa en dos casos

- e == x = true
- e == x = false

Principio de Extensionalidad sobre Listas

Dada una lista, que no sabemos que pinta tiene, podemos predicar sobre ella diciendo que tenemos dos casos

- xs = []
- $\exists z :: a, z :: [a]\ /\ xs = (z : zs)$

Demostración de Desigualdades

¿Cómo demostramos que no vale una igualdad e1 = e2 :: A?
Entiendo que básicamente hay que encontrar hacer **ALGO** para demostrar que no son iguales. Este **ALGO** es una funcionalidad que devuelva algo con uno, y otra cosa con otro. Por lo tanto debería ser del tipo $obs\ a :: A \to Bool$.
Demostrar que **no** vale la igualdad: $id = swap :: (Int, Int) \to (Int, Int)$.
Esto es fácil de probar porque sabemos que id nos da exactamente lo mismo que envíamos, y swap justamente da vuelta todo. Por lo tanto podemos comparar el primer elemento con id y el primer elemento haciendo el swap.
Ej.: $(1, 2) \to id(1, 2) \to (1, 2)$ pero $swap(1, 2) = (2, 1)$ y el primer elemento de $id(1, 2) \to (1, 2)$ es decir, $1 \neq 2$ que arroja el swap.
Por lo tanto

```
Ej: (1, 2)
obs :: ((Int, Int) -> (Int, Int)) -> Bool
obs f = fst (f (1,2)) == 1

obs id -> True
obs swap False
```

Nótese que el obs está **exactamente armado para este caso particular**. Para demostrar que las funciones no hacen lo mismo.

Isomorfismo de Tipos

Decimos que dos tipos son isomorfos si podemos pasar de uno al otro aplicando alguna función y al componerlas el resultado arroja la función identidad.

```
("hola", (1, True)) :: (String, (Int, Bool))
((True, "hola"), 1) :: ((Bool, String), Int)
Estos dos tipos son isomorfos porque podemos transformar los valores de un tipo en valores del otro

f :: (String, (Int, Bool)) -> ((Bool, String), Int)
f (s, (i, b)) = f((b, s), i)
```

```
g :: ((Bool, String), Int) -> (String, (Int, Bool))
g ((b, s), i) = f (s, (i, b))
```

¡Es básicamente crear una función que reciba los parámetros de otra manera y los mande a la otra función de la otra forma! Una especie de Curry/Uncurry.
Con esto se puede demostrar que $g.f = id$ y $f.g = id$
Formalmente, podemos decir que dos tipos de datos A y B son **isomorfos** si

- Hay una función $f :: A \to B$ total.
- Hay una función $g :: B \to A$ total
- Se puede demostrar que $g.f = id :: A \to A$
- Se puede demostrar que $f.g = id :: B \to B$

En criollo: Debe existir una función que me mapee de f a g y viceversa. Por último, si hago f(g) o g(f) tienen que dar la identidad.
Notación: $A \simeq B$ indican que A y B son isomorfos.

Intérpretes

Un intérprete ejecuta programas, no los traduce como si fuese un compilador.

Sintaxis Concreta

Se pueden representar programas como cadenas de texto. Es tedioso de interpretar algo así: if $x > 0$ then x else -x

Sintaxis Abstracta

Podemos representar instrucciones a través de árboles: (Eif (gt (var "x") cte 0)) (var("x")) (neg (var("x")))

Lenguaje de Expresiones Aritméticas

Veamos como hacer un intérprete que pueda reconozca números y haga sumas.
Primero tenemos que definir los tipos de expresiones que vamos a querer reconocer (expresiones) en nuestro intérprete.

```
data Expr = EConstNum Int
          | EAdd Expr Expr
```

Nótese que EAdd es un constructor del tipo Expr recursivo que en algún momento llegará al caso base de EConstNum que devuelve un Int.
¿Es posible comparar los resultados de las Expr desde una manera de igualdad? ¡No! Los tipos no aceptan **ningún tipo de operación** a menos que lo indiquemos. Si queremos que estos tipos puedan compararse de la siguiente forma $n_1 = n_2$ tenemos que configurar **deriving (EQ)** donde EQ es una clase.

```
data Expr = EConstNum Int
          | EAdd Expr Expr
          deriving Eq
```

¿Es posible mostrar el resultado en consola de las operaciones que hagamos? ¡No! Los tipos no aceptan **ningún tipo de operación** a menos que lo indiquemos. Si queremos que el resultado pueda mostrarse en consola de alguna manera específica tenemos que hacer una instancia de Show.

```
instance Show Expr where
show = showExpr
  where
    showExpr :: Expr -> String
    showExpr (EConstNum n) = show n
    showExpr (EAdd e1 e2) = "(" ++ show e1 ++ "+" ++ show e2 ++ ")"
```

Bueno, ahora sí. ¿Cómo podemos usar todo esto para computar nuestras expresiones? Necesitamos de alguna manera una función que nos permita enviar expresiones, tomarlas por pattern matching y hacer algo.

```
evalExpr :: Expr -> Int
evalExpr (EConstNum n) = n
evalExpr (EAdd n m) = evalExpr n + evalExpr m
```

Pero ¿cómo lo usamos?. Primero tenemos que llamar a la función enviando la información que nos pide, es decir, algo del tipo Expr.
Probemos hacer 1+2, esto en sintáxis concreta sería algo como: $evalExpr\ (EAdd\ (EConstNum\ 1)\ (EConstNum\ 2))$. ¡Nótese que estamos utilizando los constructores del tipo!
Entonces, el resultado es: $ghci >\ evalExpr\ (EAdd\ (EConstNum\ 1)\ (EConstNum\ 2)) \implies 3$

Nuestro Lenguaje de Expresiones Aritméticas con Booleanos

Queremos agregarle booleanos a nuestro lenguaje. ¿Qué debemos hacer? Lo primero es lo primero, tenemos que ver si debemos refactorizar algo o simplemente extender. Como hasta ahora hicimos solamente cosas de números raramente vamos a tener que refactorizar algo aunque eso está por verse.
Por lo tanto lo primero que debemos hacer es aceptar que el booleano sea un constructor del tipo Expr

```
data Expr =   EConstNum Int
            | EConstBool Bool
            | EAdd Expr Expr
```

Entonces ahora podemos extender la clase Show

```
    ... mismo antes
    showExpr (EConstBool b) = show b
```

Por último, podemos aceptar evaluar la expresión de booleano (que sería un caso base) ante eventuales operaciones de este tipo.

```
evalExpr :: Expr -> Int
evalExpr (EConstNum n) = n
evalExpr (EConstBool b) = b
evalExpr (EAdd n m) = evalExpr n + evalExpr m
```

¿Cuál es el problema que estamos viendo? El problema es que ahora evalExpr **no** solo devuelve Int sino que Bool. Por esto, podemos crear un nuevo tipo que sean los tipos de salida de nuestras expresiones.

```
data Val = VN Int | VB Bool  deriving Eq
```

Luego,

```
evalExpr :: Expr -> Val
evalExpr (EConstNum n) = n
evalExpr (EConstBool b) = b
evalExpr (EAdd n m) = evalExpr n + evalExpr m
```

Pero ahora hay otro problema: **evalExpr n y evalExpr m devuelven un tipo Val, el tipo Val es un Int — Bool**. Veamos para qué tipos está permitida la suma infija (Num a =¿a -¿a -¿a). Por lo tanto, **ahora que estamos usando Val no podemos usar la suma porque Val no necesariamente es un número, también puede ser un booleano**. Así que como nosotros sabemos que **vamos a usar el +** cuando tenemos números hay que hacer una función específica para ese tipo.
Creamos entonces, una función que vamos a llamar para que nos devuelva una nueva instancia del tipo Val que haga las sumas correspondientes

```
addVal :: Val -> Val -> Val
addVal (VN n) (VN m) = VN (n+m)
addVal _ _ = error "Algún sumando no es numérico"
```

Por lo tanto, reemplazamos + con addVal

```
evalExpr :: Expr -> Val
evalExpr (EConstNum n)  = VN n
evalExpr (EConstBool b) = VB b
evalExpr (EAdd n m) = addVal (evalExpr n) (evalExpr m)
```

¿Qué falta ahora? Configurar el show de nuestra nueva clase Val porque evalExpr nos devuelve algo de tipo Val y todavía no definimos cómo mostrarlo.

```
instance Show Val where
show = showVal
    where
        showVal :: Val -> String
        showVal (VN n) = show n
        showVal (VB b) = show b
```

Por último, ejecutamos igual que antes y funcionará todo: $evalExpr\ (EAdd\ (EConstNum\ 1)\ (EConstNum\ 2)) \to 3$

Nuestro Lenguaje de Expresiones con Definiciones en el Ambiente

Queremos aceptar expresiones del tipo $let\ x = 3\ in\ (let\ y = x + x\ in\ 1 + y)$
Recordemos que en Haskell, un let define una variable en un environment para que la podamos utilizar. Si nosotros llegaramos a mencionar una variable y no está definida da error.
Sabiendo esto, necesitamos dos procedimientos: definir una variable con un valor, y una función que dada una variable me de su valor.
Extendamos nuestro tipo de Expr

```
type Id = String
data Expr = EConstNum Int
            | EConstBool Bool
            | EAdd Expr Expr
            | ELet Id Expr Expr
            | EVar Id
```

Nota: Id es básicamente la forma que vamos a identificar las variables, en este caso con un String. Ej. "x"
¿Qué es lo que tenemos que refactorizar? Tenemos que aceptar los casos de EVar y ELet en nuestro evalExpr por lo tanto

```
evalExpr :: Expr -> Val
evalExpr (EConstNum n)  = VN n
evalExpr (EConstBool b) = VB b
evalExpr (ELet x e1 e2) = ?
evalExpr (EVar x) = ?
evalExpr (EAdd n m) = addVal (evalExpr n) (evalExpr m)
```

¿Cúal es el problema de esto? ¡No tenemos en el environment donde queremos almacenar la definición de la variable x!
Importante: Recordar que el Environment de Haskell inicialmente vamos a enviarlo vacío. Si llegase a haber una acción sobre él, devolvemos una instancia nueva.
Ej.: Si tenía x = 2 y ahora quiero agregar z = 4, voy a tener lo anterior + z = 4 en un nuevo ambiente. Sería una especie de asegura de especificación.
Por lo tanto, definamos **de qué manera** vamos a usar el Environment.

```

data Env a = EE [(Id, a)]

emptyEnv :: Env a
emptyEnv = EE []

lookupEnv :: Env a -> Id -> a
lookupEnv (EE e) x =
case lookup x e of
    Just y  -> y
    Nothing -> error ("La␣variable␣" ++ x ++ "␣no␣está␣definida.")

extendEnv :: Env a -> Id -> a -> Env a
extendEnv (EE e) x a = EE ((x, a) : e)

data Val = VN Int | VB Bool | VS String deriving Eq -> Agregamos VS String (para el ID)

instance Show Val where
show = showVal
    where
        showVal :: Val -> String
        showVal (VN n) = show n
        showVal (VB b) = show b
        showVal (VS s) = show s -> Agregamos
```

Ahora sí, extendamos nuestro evalExpr

```
evalExpr :: Expr -> Env Val -> Val
evalExpr (EConstNum n) _  = VN n
evalExpr (EConstBool b) _ = VB b
evalExpr (EAdd n m) env = addVal (evalExpr n env) (evalExpr m env)
evalExpr (EVar x) env = lookupEnv env x
evalExpr (ELet x e1 e2) env = ?
```

¿Cómo definimos x con la expresión 1 y la expresión 2?
Recordemos nuevamente, que acá tenemos que ir pisando el environment, que x es una expresión y en el environment tenemos que guardar su **valor** por lo tanto no hay que olvidarnos de obtener el valor de x.

```
    evalExpr (ELet x e1 e2) env =
                let v = evalExpr e1 env
                    env' = extendEnv env x v
                in evalExpr e2 env'
```

Luego, agregamos el show a la aplicación del let y el var

```
    ...lo anterior
    showExpr (ELet x e1 e2) = "let " ++ x ++ " = " ++ show e1 ++ " in " ++ show e2
    showExpr (EVar x) = x
```

Ahora, ejecutemos alguna instrucción que haya funcionado antes
$evalExpr\ (EAdd\ (EConstNum\ 1)\ (EConstNum\ 2))\ emptyEnv \rightarrow 3$
Probemos ahora sí ejecutar expresiones del tipo let y var. Definamos una variable x = 3, hagamos y = 4 y luego x + y

- Sintaxis
  - Input: (ELet "x" (EConstNum 3) (ELet "y" (EConstNum 4) (EAdd (EVar "x") (EVar "y"))))
  - Output Esperado: let x = 3 in let y = 4 in (x+y)
- Código
  - Input: evalExpr (ELet "x" (EConstNum 3) (ELet "y" (EConstNum 4) (EAdd (EVar "x") (EVar "y")))) emptyEnv
  - Output: 7

Simulando un Lenguaje Imperativo con nuestro Intérprete

¿Qué es lo que tiene un lenguaje imperativo que no tiene un lenguaje funcional? la asignación.
La asignación en un lenguaje imperativo se hace comúmente de la siguiente forma

- El nombre de la variable se busca su address en memoria.
- Se hace una escritura sobre ese address con un nuevo valor manteniendo el mismo nombre.

Entonces de alguna manera necesitamos simular una memoria. Para eso definamos el tipo y las funciones que necesitamos (los dió la catedra).

```
    module Memory(Addr, Mem, emptyMem, freeAddress, load, store) where

    type Addr = Int
    data Mem a = MM [(Addr, a)]

    instance Show (Mem a) where
    show (MM _) = "<memoria>"

    emptyMem :: Mem a
    emptyMem = MM []

    freeAddress :: Mem a -> Addr
    freeAddress (MM []) = 0
    freeAddress (MM xs) = maximum (map fst xs) + 1

    load :: Mem a -> Addr -> a
    load (MM xs) a =
    case lookup a xs of
        Just b  -> b
        Nothing -> error "La dirección de memoria no está."

    store :: Mem a -> Addr -> a -> Mem a
    store (MM xs) a b = MM ((a, b) : xs)
```

Luego, modificamos las acciones que pueden tener nuestras expresiones en un lenguaje imperativo: asignar un valor nuevo a una variable y ejecutar una secuencia de expresiones

```
type Id = String
data Expr = EConstNum Int
          | EConstBool Bool
          | EAdd Expr Expr
          | ELet Id Expr Expr
          | EVar Id
          | EAssignID Expr
          | ESeq Expr Expr
```

Entonces ahora sí ¿qué cambia de evalExpr?

- Ya no recibimos el valor por una Env sino que ahora recibimos un address.
- El valor de un address particular está atado a la memoria y depende de esa memoria.

```
evalExpr :: Expr -> Env Addr -> Mem Val -> (Val, Mem Val)
evalExpr (EVar x) env mem = (load mem(lookupEnv x), mem)
evalExpr (EAdd e1 e2) env mem =
        let (v1, mem') = evalExpr e1 env mem
            let (v2, mem') = evalExpr e2 env mem'
            in (addVal v1 v2, mem'')
    evalExpr (ELet x e1 e2) env mem =
        let addr = freeAddr mem
            (val, mem') = evalExpr e1 env mem
                mem'' = store mem' addr v
            in evalExpr e2 env' mem''
evalExpr (EAssign x expr) env mem =
        let addr = lookupEnv env x
        (v, mem') = evalExpr e env mem
            mem'' = store mem' addr v
        in (v, mem'')

evalExpr (ESeq e1 e2) env mem =
    let (v1, mem') = evalExpr e1 env mem
        in eval e2 env mem'
```

Agregando los condicionales a nuestro intérprete

```
evalExpr (EIf c t e) env mem =
    case eval c env mem of
        (VB b, m) -> evalExpr (if b then t else e) env m
        (VN n, m) -> error "Cond no bool"
```

Nota: Es posible encontrar este código en la carpeta de teóricas hecho por mí.

Características Funcionales

Volvamos a nuestro lenguaje púramente funcional sin memoria ni direcciones.

```
evalExpr :: Expr -> Env Val -> Val
evalExpr (EConsNum n) _ = h
evalExpr (EConsBool b) _ = b
evalExpr (EAdd e1 e2) _ = evalExpr e1 env 'addVal' evalExpr e2 env
evalExpr (EVar x) env = lookup env x
evalExpr (ELet x e1 e2) env = evalExpr e2 (extendEnv env x (evalExpr x env))
evalExpr (EIf c t e) env = evalExpr (if isTrue(evalExpr c env) then t else e) env
    where isTrue (v b x) = x
          isTrue _ = False
```

Prácticamente todos los lengaujes funcionales están basados en el cálculo-λ
El cálculo-λ es un lenguaje que tiene solamente tres constructores.

- EVar Id $--x$

- ELam Id Expr $--\backslash x \to e$
- EApp Expr Expr $--e1\ e2$

Ahora queremos evaluar cosas como $(\backslash x \to x + x)$
Por lo tanto en nuestro data Val agreguemos el constructor VFunction, y en las expresiones que queremos evaluar agreguemos la composición y las funciones lambda.

```
data Val = VN Int | VB Bool | VFunction Id Expr

data Expr = ...
            | ELan Id expr -- \x -> c
            | EApp expr expr -- e1(e2)
```

Ahora agreguemos el cuerpo de esas operaciones de expresiones

```
...
evalExpr (ELam x e) env = VFunction x e
evalExpr (EApp e1 e2) env =
    let v1 = evalExpr e1 env
    let v2 = evalExpr e2 env
    in case v1 of
        VFunction x e -> evalExpr e (extendEnv env x v2)
                    _ -> error "aplicando una no funcion"
```

Importante: la e es el cuerpo de la función mientras que (extendEnv env x v2) es el pasaje de parámetros a una función lambda. Los parámetros se pasan a través del environment.
Con esta implementación tenemos un problema, si queremos hacer lo siguiente

```
let suma = \x -> \ y -> x + y in
let f = suma 5 in
let x = 0 in
    f 3
```

Tenemos el problema que nosotros querríamos que eso sea 8, pues estamos justamente aplicando parcialmente suma enviando un 5 pero esto da 3. ¿Por qué?
En Haskell, este comportamiento no sucede (porque ya lo contempla) pero es importante entender **qué es lo que pasa**.
Cuando hacemos let f = suma 5, f es un nombre a la función parcialmente aplicada suma que como primer argumento se le envía 5.
Sin embargo, cuando hacemos x = 0 f 3 lo que estamos haciendo es pisar el comportamiento de f diciendo ahora que la x que antes habíamos indicado vía (suma 5) ya no es suma 5, sino que ahora es suma 0.
Entonces, el paso a paso sería algo así

```
let suma = \x -> \y -> x + y in
let f = suma 5 in

f = \y -> x + y -> f no recuerda nada de lo definido parcialmente

let x = 0 in
    f 3

f = \3 -> x + 3 -> x tiene el valor de 0 en el entorno
f = 0 + 3 = 3
```

Importante: Las funciones parcialmente aplicadas deben recordar la definición en el ambiente que se les dió y qué parametros se les dieron a la hora de definirlas. De lo contrario, podría cambiar su comportamiento. Esto es conocido como **Closures**

Closures

Son funciones que recuerdan su ambiente a la hora de ser definidas. Cuando decimos ambiente decimos también con qué argumentos se definieron. Se utiliza para **capturar el entorno** en el que la función fue creada, permitiendo que acceda a variables externas en su ámbito local.
En lenguajes como JavaScript se ven de la siguiente forma

```
function crearContador() {
let contador = 0;
return function() {
```

```
        contador++;
        return contador;
    }
}

const contador1 = crearContador();
console.log(contador1()); // 1
console.log(contador1()); // 2
```

Importante: Un closure mantiene la referencia al entorno donde fue creada, permitiendo que acceda a las variables de ese entorno en ese momento incluso después de que el entorno ya no exista. En nuestro intérprete debemos reemplazar las VFunctions por VClosures.
Los Closures suelen estar ligados a estrategias de evaluación, es decir, a las técnicas para evaluar aplicaciones de dos funciones.

Thunk

Un Thunk es una función que **no se evalúa hasta que realmente se lo necesite**. Es una técnica para diferir la evaluación de una expresión hasta que su valor sea necesario.
Ej.: let exp $= 2 + 3$ no devuelve 5 hasta que realmente se lo necesite.
En lenguajes como JavaScript se ven de la siguiente forma: let thunk $= () \implies 2 + 3$; entonces cuando la necesitamos hacemos básicamente thunk()
Importante: Un Thunk difiere la evaluación de una expresión pero no capturan variables de su entorno.

Estrategias de Evaluación de dos funciones

- Llamada por valor (call-by-value)
  - Se evalúa e1 hasta que sea un closure.
  - Se evalúa e2 hasta que sea un valor.
  - Se evalúa el cuerpo de la función utilizando el resultado de e2 (valor)
  - El parámetro queda ligado al valor de e2.
- Llamada por nombre (call-by-name)
  - Se evalúa e1 hasta que sea un closure.
  - Se evalúa el cuerpo de la función.
  - El parámetro queda ligado a la expresión e2 sin evaluar (en este contexto, es un thunk)
  - **Cada vez** que se usa el parámetro se evalúa la expresión e2.
- Llamada por necesidad (call-by-need)
  - Se evalúa e1 hasta que sea un closure..
  - Se evalúa el cuerpo de la función.
  - El parámetro queda ligado a la expresión e2 sin evaluar.
  - **La primera vez** que el parámetro se necesita, se evalúa e2.
  - Se guarda el resultado para evitar evaluar e2 nuevamente (necesito memoria)

Importante: En colores, están los pasos que comparten entre las diferentes estrategias.

Aplicando Closures a nuestro Intérprete (Call By Value)

Cambiemos entonces, VFunction por VClosures.

```
    data Val = VN Int | VB Bool | VClosure Id Expr (Env Val)
```

¡Importante el env! Es la idea de los closures.
Ahora modifiquemos el evalExpr para utilizar closures.

```
    evalExpr :: Expr -> Env Val -> Val
    evalExpr (ELam x e) env = VClosure x e env
    evalExpr (EApp e1 e2) env =
        let v1 = eval e1 env
        let v2 = eval e2 env
            in case v1 of
                VClosure x e env' -> evalExpr e (extendEnv env' x v2)
                                _ -> error "Aplicando una no función"
```

¿Por qué esta estrategia es Call By Value? Porque al evaluar al expresión e estamos directamente pasando el valor de evaluar e2 de antemano.

Aplicando Closures a nuestro Intérprete (Call By Name)

Como acá necesitamos sí o sí diferir de no calcular el valor de e2 de antemano, entonces acá **evaluamos expresiones no valores**.

```
data Thunk = TT Expr (Env Thunk)
data Val = ...
         | VClosure Id Expr (Env Thunk)

evalExpr :: Expr -> Env Thunk -> Val
evalExpr (EVar x) env =
                case lookup env x f
                tt e env' -> eval e env'
evalExpr (EApp e1 e2) env =
    let v1 = eval e1 env
        in case v1 of
            VClosure x e env' -> evalExpr e (extendEnv env' x) (tt e2 env)
                        _ -> error "Aplicando una no función"
```

¿Por qué esta estrategia es Call By Name? Porque en ningún momento evalúamos e2 si no la necesitamos. Sino que solamente se evalúa obteniendo su valor cuando se necesita en su ambiente dado.

Aplicando Closures a nuestro Intérprete (Call By Need)

En call-by-need hay dos tipos de valores

- Valores que ya están calculados (atómicos): enteros, booleanos, closures.
- Valores pendientes de ser evalúados (thunks).

Para utilizar call by need tenemos que tener en cuenta

- Asociación de Identificadores a Direcciones: Esto quiere decir que el entorno (Env) asocia los nombres de las variables (identificadores) con **direcciones de memoria**.
- Asociación de Direcciones a Valores: Esas direcciones de memoria de los identificadores tienen valores finales o thunks.
- Evaluar una expresión: Nos da como resultado su valor final.

Ej conceptual: let x = (2 + 3) * 4
En este ejemplo x es básicamente un nombre o identificador que almacena un thunk dentro (2+3) * 4.
Si en algun momento se llegase a utilizar, x toma el valor de 20 y lo guarda en la memoria.

Sistemas Deductivos

Es una herramienta matemática que formaliza el lenguaje de la demostración.
Existen muchos sistemas deductivos, acá vamos a ver el de deducción natural que es uno parecido a la forma de pensar del humano.
Está dado por un conjunto de **axiomas** y **reglas de inferencia** que tienen la siguiente estructura

$$\frac{}{\text{⟨ axioma ⟩}}\ \text{⟨ nombre ⟩} \qquad \frac{\langle premisa_0\rangle\ \langle premisa_1\rangle ... \langle premisa_n\rangle}{\langle conclusion\rangle}\ \text{⟨ nombre de la regla ⟩}$$

Axioma: Afirmaciones básicas que se asumen como verdaderas. No es posible deducirlas de otras afirmaciones.
Reglas de Inferencia: Permiten derivar afirmaciones (teoremas) a partir de axiomas y otras afirmaciones.

Árbol de Derivación

- Las premisas son hojas.
- Los nodos representan afirmaciones.
- La raíz es la afirmación que se quiere probar.
- Las ramas representan las reglas de inferencias que conectan a las afirmaciones.

Si llegamos a una prueba correcta, **las hojas son axiomas**.

Afirmación Derivable (Teorema)

Una afirmación es derivable si existe alguna derivación sin premisas que la tiene como conclusión.

Fórmulas

- Cualquier variable proposicional es una fórmula.
- Si ρ es una fórmula, entonces $\neg\rho$ es una fórmula.
- Si ρ y σ son fórmulas, $\rho \wedge \sigma$ es una fórmula.
- Si ρ y σ son fórmulas, $\rho \vee \sigma$ es una fórmula.
- Si ρ y σ son fórmulas, $\rho \implies \sigma$ es una fórmula.
- Si ρ y σ son fórmulas, $\rho \iff \sigma$ es una fórmula.

Al ser un conjunto inductivo, viene provisto de

- Esquema de prueba para probar propiedades sobre ellos **inducción estructural**.
- Esquema de recursión para definir funciones sobre el conjunto **recursión estructural**.

Gramática de la Lógica Proposicional

Las fórmulas son las expresiones que se pueden generar a partir de la siguiente gramática

$$\tau, \sigma, \rho... ::= P \mid \bot \mid (\tau \wedge \sigma) \mid (\tau \implies \sigma) \mid (\tau \vee \sigma) \mid \neg\tau$$

Llamamos **contexto** a un conjunto finito de fórmulas.

Valuación

Una valuación es una función $v : \mathcal{V} \implies \{V, F\}$ que asigna valores de verdad a las variables proposicionales.
Una valuación satisface una propositión τ si $v \vDash \tau$ donde:

- $v \vDash P \; sii \; v(P) = V$
- $v \vDash \neg\tau \; sii \; v \nvDash \tau$
- $v \vDash \tau \vee \sigma \; sii \; v \vDash \tau \; o \; v \vDash \sigma$
- $v \vDash \tau \wedge \sigma \; sii \; v \vDash \tau \; y \; v \vDash \sigma$
- $v \vDash \tau \implies \sigma \; sii \; v \nvDash \tau \; o \; v \vDash \sigma$
- $v \vDash \tau \iff \sigma \; sii \; v \vDash \tau \; sii \; v \vDash \sigma$

Nota: Una valuación es una fila de la tabla de verdad.
Nota 2: $\vDash$ se lee como **satisface**
A modo de ejercicio escribamos uno de estas valuaciones con árboles de derivación a nivel de sistema deductivo:

$$\frac{v \vDash \tau \; v \vDash \sigma}{\langle \; v \vDash \tau \wedge \sigma \; \rangle} \quad \langle \wedge \rangle$$

Equivalencia Lógica y Tipos de Fórmulas

Dadas dos fórmulas τ y σ:

- τ es lógicamente equivalente a σ cuando $v \vDash \tau$ sii $v \vDash \sigma$ para toda valuación v.

Una fórmula τ es:

- Una tautología si $v \vDash \tau$ para toda valuación v.
- Satisfactible si existe una valuación v tal que $v \vDash \tau$
- Insatisfactible si no es satisfactible.

Un conjunto de fórmulas Γ es

- Satisfactible si existe una valuación v tal que $\forall \tau \in \Gamma$ se tiene $v \vDash \tau$
- Insatisfactible si no es satisfactible.

Teorema de la Insatisfactibilidad

Una fórmula τ es una tautología sii $\neg\tau$ es insatisfactible.

Sistema Deductivo basado en Reglas de Prueba

- Secuente (expresión que incluye conclusión): $\tau_1, \tau_2, ..., \tau_n \vDash \sigma$
  - Denota que a partir de asumir que el conjunto de fórmulas $\tau_1, \tau_2, ..., \tau_n$ son tautologías, podemos obtener una prueba de la validez de σ
- Reglas de prueba:

$$\frac{\Gamma_1 \vDash \tau_1 \dots \Gamma_n \vDash \tau_n}{\Gamma \vDash \sigma : conclusion}$$ nombre de la regla

Convenciones de Notación

- Omitimos los paréntesis más externos de las fórmulas: $\tau \wedge \neg(\sigma \vee \rho) = (\tau \wedge \neg(\sigma \vee \rho))$
- La implicación es asociativa a la derecha: $\tau \implies \sigma \implies \rho = (\tau \implies (\sigma \implies \rho))$
- Los conectivos $\wedge, \vee$ **no** son conmutativos ni asociativos
  - $\tau \vee (\sigma \vee \rho) \neq (\tau \vee \sigma) \vee \rho$
  - $\tau \wedge \sigma \neq \sigma \wedge \tau$

Consecuencia Lógica

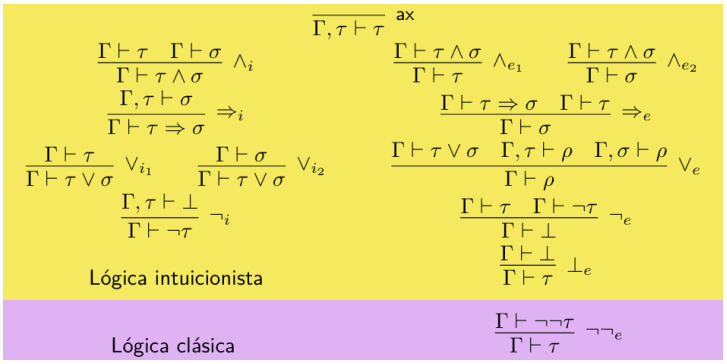
- $\bot$ es siempre falso
- $v \vDash \emptyset$
- $\emptyset \vDash T$ todas las tautologías

Lógica Intusionista (NJ) vs Lógica Clásica (NK)

Las reglas de la lógica intusionista están contenidas en las reglas de la lógica clásica.
Algunas reglas que podemos aplicar en la **lógica clásica** que no podemos usar en la Lógica Intusionista

- PBC (proof by contradiction): No es derivable, es decir, no se puede deducir.
- LEM (law of excluded middle): Es un axioma, no tiene premisas y por lo tanto no se puede probar.
- $\neg\neg e$ (doble negation elimination)

Las reglas **PBC**, **LEM** y $\neg\neg e$ son equivalentes.



Reglas intuicionistas

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \neg\neg\tau}\ \neg\neg_i \qquad \frac{\Gamma \vdash \tau \Rightarrow \sigma \quad \Gamma \vdash \neg\sigma}{\Gamma \vdash \neg\tau}\ \text{MT}$$

Reglas clásicas

$$\frac{\Gamma, \neg\tau \vdash \bot}{\Gamma \vdash \tau}\ \text{PBC} \qquad \frac{}{\Gamma \vdash \tau \vee \neg\tau}\ \text{LEM}$$

Weakening (Debilitamiento) y Strengthening (Fortalecimiento)

Weakening: Lo voy a decir en criollo, su demostración se hace por inducción, pero básicamente es que si algo se cumple solo con τ entonces si agrego otras hipótesis también seguirá valiendo.
Se dice que es Weakening porque estamos haciendo más débil la prueba, esto es porque a mayor cantidad de información que tiene el contexto es más dificil de probar.

$$\frac{}{\Gamma \vDash \sigma} \implies \frac{}{\Gamma, \tau \vDash \sigma}$$

Strengthening: Es lo contrario a Weakening, la idea es que podemos ir sacando hipótesis que no nos sirven y poder seguir probando que vale. Se utiliza mucho en algo llamado Cálculo Lambda que veremos luego.
Se dice que es Strengthening porque estamos haciendo más fuerte la prueba, esto es porque a menor cantidad de información que tiene el contexto es más fácil de probar.

Correctitud (Soundness) y Completitud

Importante: Utilizamos $\vDash$ para hablar de satisfacibilidad en el contexto de **semántica** mientras que utilizamos $\vdash$ para indicar que una fórmula se puede derivar en el contexto de **sintáxis**.

Correctitud y Completitud responde a la siguiente pregunta: ¿cuál es la relación entre la sintáxis y la semántica?

- **Sintáxis**: Conjunto de fórmulas τ tal que $\neg\tau$ es un secuente válido.
- **Semántica**: Conjutno de fórmulas τ tal que $v \vDash \tau$ para toda valuación v. Ej: Tautologías.

$$\Gamma \vDash T \iff \Gamma \vdash T$$

Corrección
$\vDash \tau$ secuente válido implica que τ es tautología.

- $\sigma_1, \sigma_2, \ldots, \sigma_n \vdash \tau$ secuente válido implica que $\sigma_1, \sigma_2, \ldots, \sigma_n \vDash \tau$

Para demostrar: Hacemos inducción en la estructura de la prueba analizando por casos la última regla aplicada en la prueba.
Caso base, la última regla fue un axioma.
Casos recursivos, el resto de reglas $\wedge_i, \wedge_{e1}, etc$
Completitud τ tautología implica que $\vdash \tau$ es secuente válido.

- $\sigma_1, \sigma_2, \ldots, \sigma_n \vDash \tau$ implica que $\sigma_1, \sigma_2, \ldots, \sigma_n \vdash \tau$ es un secuente válido.

Para demostrar: Usamos el contrarrecíproco: si $P \implies Q$ vale que $\neg Q \implies \neg P$
Luego probar estos dos lemas

- Si $\Gamma \nvdash \tau$ entonces $\Gamma \cup \{\neg\tau\}$ es consistente (sale por contrarrecíproco).
- Si Γ es consistente, entonces tiene modelo. Ej.: Γ es satisfactible (es más dificil).

Consecuencia Semántica

Sean $\tau_1, \tau_2, \ldots, \tau_n, \sigma$ fórmulas de la lógica proposicional.
$\tau_1, \tau_2, \ldots, \tau_n \vDash \sigma \iff \forall\ v\ valuacion\ ((v \vDash \tau_1 \wedge v \vDash \tau_2 \wedge \cdots \wedge v \vDash \tau_n) \implies (v \vDash \sigma))$
En criollo: Vale que $\tau_1, \tau_2, \ldots, \tau_n \vDash \sigma \iff$ para toda valuación v tal que v satisface a todas las hipótesis $v \vDash \tau_i$ para toda $i \in 1, \ldots, n \implies v \vDash \sigma$

Consistencia

Decimos que Γ es consistente si $\Gamma \nvdash \bot$, es decir, Γ es consistente si no se puede derivar una contradicción a partir de él.

Cálculo Lambda (λ)

Con booleanos

Definamos algunas cosas

- **Tipos**: $\tau, \sigma, \rho, \cdots ::= bool \mid \tau \to \sigma$. Asumimos que el constructor de tipos $\leftarrow$ es asociativo a derecha.
- **Términos**: Variables $X = \{X, Y, Z, \dots\}$
- **Construcción explícita de función (sin nombre)**: $\lambda x.M$
- **Aplicación de una función a un argumento**: MN
- **Constantes para cada valor de verdad**: true y false.

Observación: Todo es una función, es decir, una función tomando un argumento constante u otra función es lo mismo.
Ej.: Una función que toma una función y la compone consigo misma es $\lambda f.\lambda x.f(fx)$
Observación 2: No existen funciones que tomen varios argumentos.
Ej.: $f(x, y) =$ if y then x else true en cálculo lambda se escribe $\lambda y.\lambda x.if\ x\ then\ y\ else\ true$
Observación 3: La aplicación asocia a la izquierda: $F\ false\ true = (F\ false)\ true$
Observación 4: La abstracción y el if tienen menor precedencia que la aplicación.
Ej.: $\lambda x : \tau.M\ N = \lambda x : \tau.(M\ N) \neq (\lambda x : \tau.M)N$
Observación 5: Las implicaciones asocian a la derecha $A \to B \to C = A \to (B \to C)$

Gramática del cálculo lambda con booleanos

Los términos válidos de cálculo lambda con booleanos los podemos describir con una gramática formal

$$M\ ::=\ x \mid \lambda x\ .\ M \mid MM \mid true \mid false \mid if\ M\ then\ M\ else\ M$$

0.1. Tipos en Cálculo Lambda

En el Cálculo Lambda tipado damos los tipos en las operaciones.
Ej.: $\backslash \lambda x : \sigma\ .\ M$

Función Identidad

En cálculo lambda existe una función id **propia por cada tipo**.

0.2. Semántica Operacional

La **gramática** nos dice qué terminos podemos escribir sintácticamente.
La **semántica** nos da el significado de los términos.
Las siguientes reglas tienen la forma $M \to N$ y se lee $M\ reduce\ a\ N$ o $M\ se\ reescribe\ como\ N$

- $(\lambda x.M)N \to M\{x := N\}$ (β reducción)
- if true then M else $N \to M$ (if_t)
- if false then M else $N \to N$ (if_f)

0.2.1. Captura de variables

$\lambda x : bool\ .\ x \equiv$ boolean main(boolean var)$\{return\ x\}$
Cuando las variables están ligadas a un cuantificador podemos cambiarle el nombre sin problema alguno (pensar en el nombre de un parámetro de función en un lenguaje imperativo).
Ej.: $x\ (\lambda x : bool\ .\ x) \equiv x\ (\lambda y : bool\ .\ x)$
Esto de que cambiar el nombre de la variable ligada al cuantificador se debe a que son clases de equivalencia (α).

0.2.2. Conjunto de Variables Libres

Se denota como $fv(M)$.

0.3. Sistema de Tipos

Hasta ahora veníamos utilizando tipos estrictos, pero normalmente el Cálculo Lambda **no viene tipado**.
Los tipos de cálculo lambda con booleanos los definimos inductivamente por

- Bool es un tipo.
- Si τ y σ son tipos, $\tau \to \sigma$ es un tipo que representa las funciones de τ en σ.

Gracias a los tipos, debemos escribir **de qué tipo son cada una de las variables**.
Importante: Solo nos vamos a interesar en **términos sin variables libres**, es suficiente con escribir el tipo cuando se liga la variable.
Nota: Solo los subtérminos tendrán variables libres.
Ej.: $\lambda x : Bool.x$ es la identidad sobre los booleanos.
Ej.: $\lambda x : Bool \to Bool.x$ es la identidad sobre las funciones de booleanos en booleanos.

La gramática de cálculo lambda con booleanos tipado la definimos con una **gramática de tipos** y una de **términos** de la siguiente manera

- $\tau ::= Bool \mid \tau \to \tau$
  - Nota: $\tau \to \tau$ representa una función.
- $M ::= x \mid \lambda x : \tau.M \mid MM \mid tt \mid ff \mid if\ M\ then\ M\ else\ M$

Donde M representa un término y τ el tipo.

0.4. Contexto

Dijimos anteriormente que solo damos tipo a los **términos sin variables libres** pero ¿que pasa con este caso: $\lambda x : Bool.yx$?
Podemos notar que y es una variable libre, y como estamos usando tipos booleanos tiene que ser una función Bool en algo pero ¿como defino ese algo?
Un **contexto** nos da tipos para variables, entonces en vez de decir $\lambda x : Bool.yx : Bool \to Bool$ decimos, si $y : Bool \to Bool$, entonces $\lambda x : Bool.yx : Bool \to Bool$
Notación: $y : Bool \to Bool \vdash \lambda x : Bool.yx : Bool \to Bool$ donde del lado izquierdo del $\vdash$ definimos el contexto.

0.4.1. Contexto de Tipado

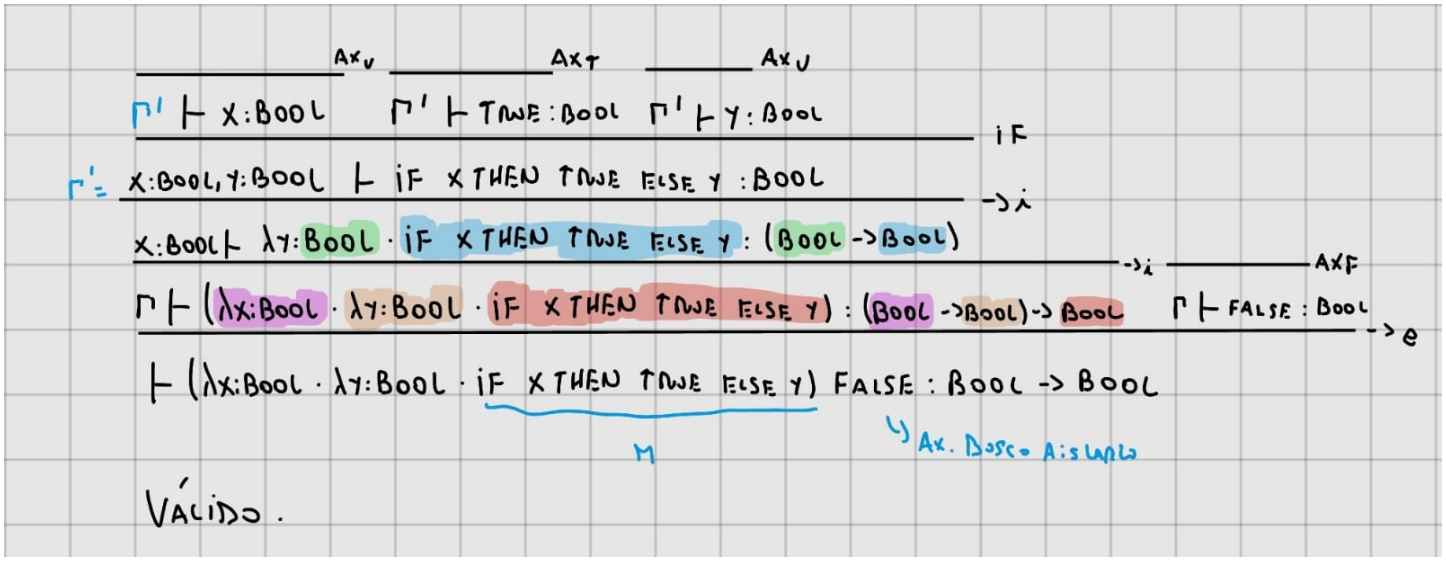
Es un conjunto finito de pares $(x_i : \tau_i)$

$$\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$$

Nota: Lo notamos con letras griegas mayúsculas $(\Gamma, \triangle)$ y **sin variables repetidas**, es decir, $(i \neq j \implies x_i \neq x_j)$

0.4.2. Juicios de Tipado

El sistema de tipos predica sobre **juicios de tipado** de la forma: $\Gamma \vdash M : \tau$
Se lee como "en el contexto Γ, M es de tipo τ"



0.5. Sistema de tipos

La relación de tipado $\Gamma \vdash M : \tau$ se define inductivamente por:

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\, ax_v \qquad \frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x{:}\tau.M : \tau \to \sigma} \to_i \qquad \frac{\Gamma \vdash M : \tau \to \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \sigma} \to_e$$

$$\frac{}{\Gamma \vdash \mathtt{tt} : \mathtt{Bool}}\, ax_t \qquad \frac{}{\Gamma \vdash \mathtt{ff} : \mathtt{Bool}}\, ax_f \qquad \frac{\Gamma \vdash M : \mathtt{Bool} \quad \Gamma \vdash N_1 : \tau \quad \Gamma \vdash N_2 : \tau}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N_1\ \mathsf{else}\ N_2 : \tau}\, \mathtt{if}$$

donde

- ax_v: variable
- ax_t: true
- ax_f: false
- $\to_i$: abstracción
- $\to_e$: aplicación
- if: condicional

Importante: Es posible que también se utilice la siguiente notación.

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}\,\text{T-TRUE} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\,\text{T-FALSE}$$

$$\frac{\Gamma \vdash M : \mathsf{bool} \quad \Gamma \vdash N : \tau \quad \Gamma \vdash P : \tau}{\Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P : \tau}\,\text{T-IF}$$

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}\,\text{T-VAR} \qquad \frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x : \tau.\, M : \tau \to \sigma}\,\text{T-ABS}$$

$$\frac{\Gamma \vdash M : \tau \to \sigma \quad \Gamma \vdash N : \tau}{\Gamma \vdash M\, N : \tau}\,\text{T-APP}$$

0.5.1. Árbol Sintático

Es muy útil hacerlo.
Véase **anexo** para ver ejemplos para reducir expresiones y verificar si son términos.

0.5.2. Término Bueno

Decimos que un término es bueno si existe un contexto del que puedo derivarlo.

0.5.3. Término Válido

Está atado a como está sintácticamente formado. Es importante que solo tengan tipos válidos.

- xM: No es válido. No podemos usar genéricos como M.
- $\lambda x : Bool$: No es válido. Las funciones toman algo y devuelven algo. Esta lambda no es $\tau \to \tau$.
- $\lambda x.x$: No es válido. No sabemos el tipo de x.

- if x then $\lambda x : Bool x$: No es válido, los condicionales tienen una rama else.
- $\lambda x : Bool \to Bool.xy\ \lambda y : Bool.x$
  - Si colocamos los paréntesis de la asociación queda así: $((\lambda x : Bool \to Bool.x\mathbf{y})\ (\lambda y : Bool.\mathbf{x}))$
  - Por un lado se resuelve $(\lambda x : Bool \to Bool.xy)$ y por otro $(\lambda y : Bool.x)$.
  - Finalmente, cuando reduce, se realiza la aplicación. En negrita, están marcadas las variables libres.
  - Este término es válido.
- $\lambda y : \sigma.y$: No es válido, no existe el tipo σ. Solo existe $Bool \mid \tau \implies \tau$
- $xy\ \lambda x : Bool \to Bool.xy$
  - Asociemos hacia la izquierda entonces queda $(xy)\ (\lambda x : Bool \to Bool.xy)$
  - Cada lado se resuelve por separado y luego se componen.

Para ver ejemplos de esto utilizando árboles sintáticos véase **anexo**

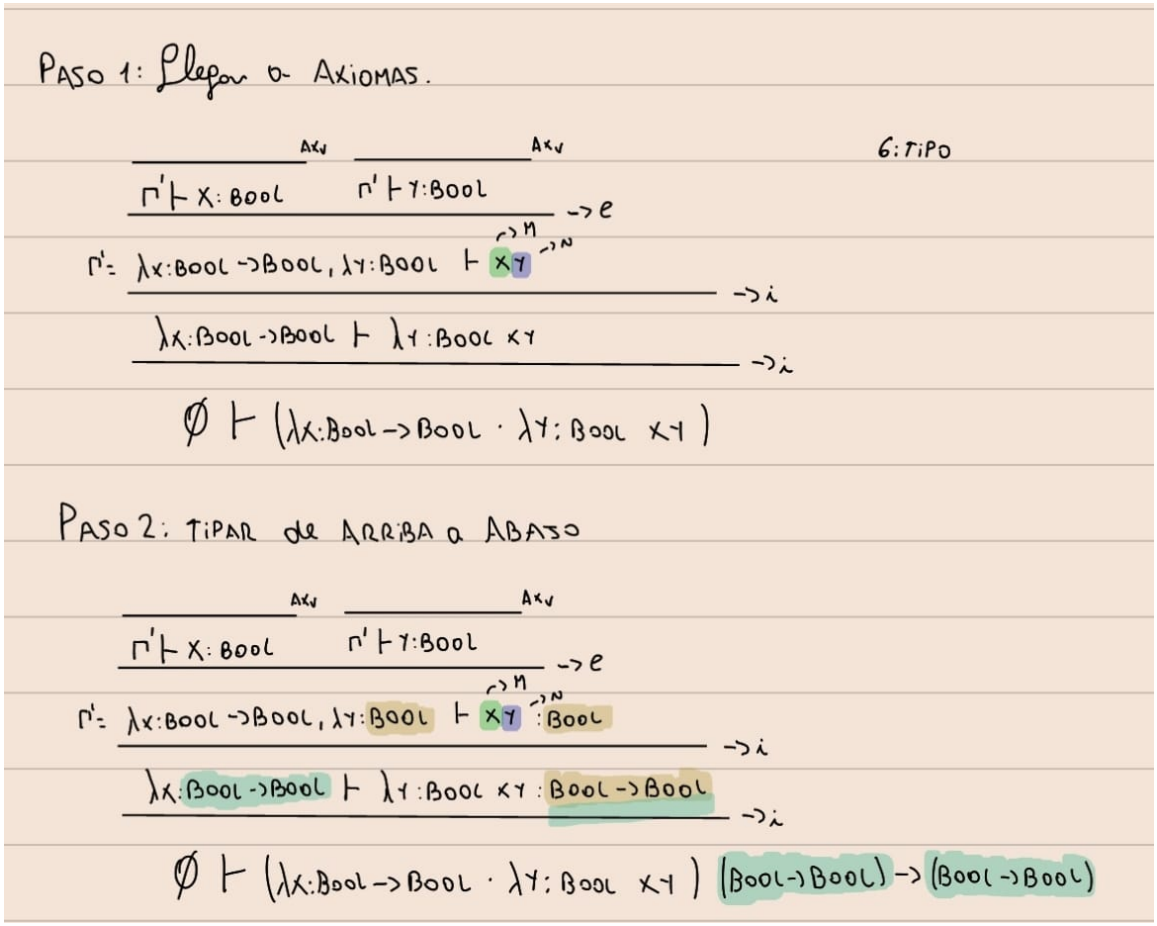
0.5.4. Término Cerrado

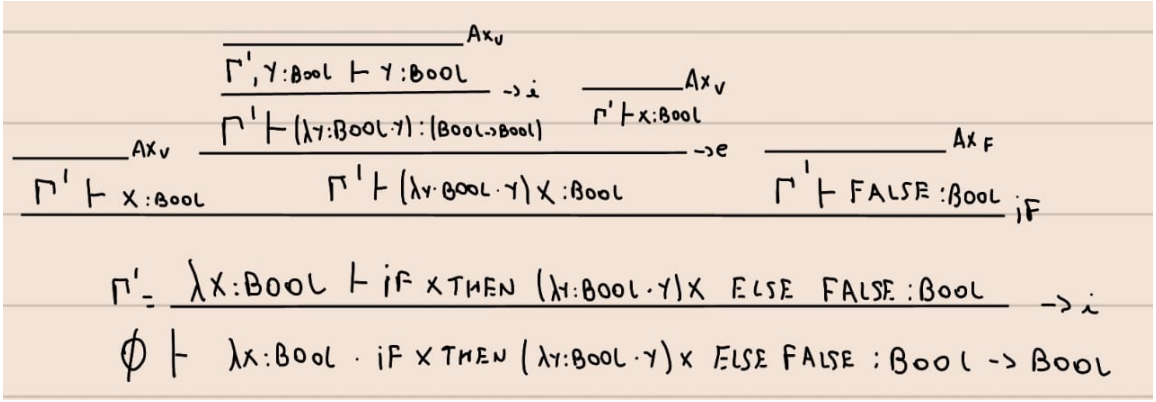
Decimos que un término es cerrado sí y solo sí no tiene variables libres.

0.5.5. Contexto Vacio

Un contexto está vacío si no existe ninguna variable libre.

0.5.6. Tipando operaciones



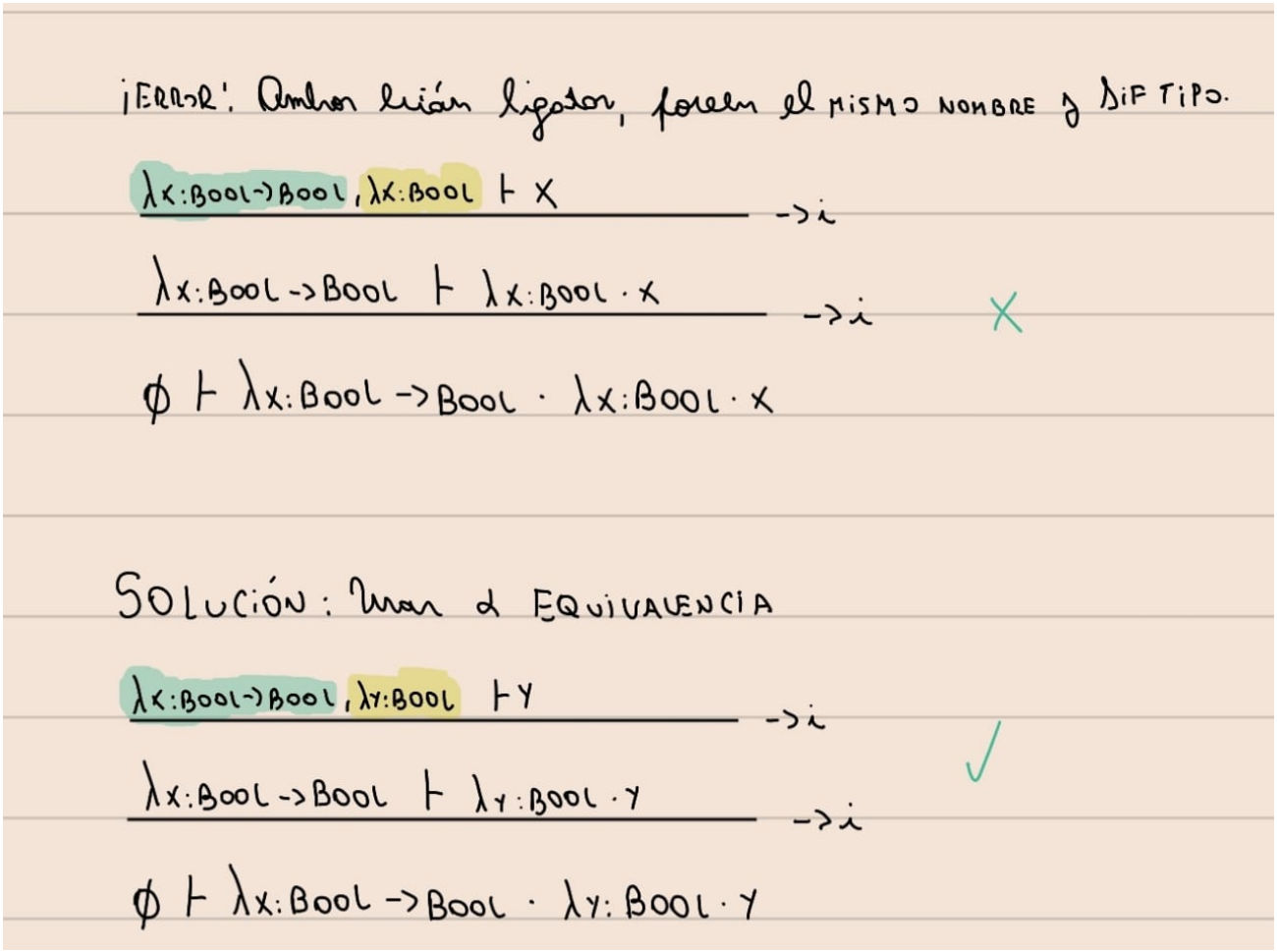


0.5.7. Clases de Equivalencia (Alfa equivalencia)

Los términos que difieren sólo en el nombre de las variables $ligadas$ se consideran iguales: $\lambda x : bool\ .\ x \underset{\alpha}{\equiv} \lambda y : bool\ xy$

0.5.8. Aplicando Clases de Equivalencia

Recordemos que las variables ligadas, en cualquier lenguaje de programación serían como el parámetro de una función y da igual que nombre le pongamos. Lo importante es NO tener dos parámetros con el mismo nombre.



Nota: Al igual que en un lenguaje de programación da igual a qué variable ligada le cambiemos su nombre.

0.5.9. Unicidad de Tipos

Si $\Gamma \vdash M : \sigma$ y $\Gamma \vdash M : \rho$ ent $\sigma = \rho$

0.5.10. Teorema (Weakening + Strengthening)

Si $\Gamma \vdash M : \tau$ es derivable y $fv(M) \subseteq \Gamma \cap \Gamma'$ entonces $\Gamma' \vdash M : \tau$ es derivable

- Weakening: Agregar más hipótesis hace que la afirmación sea más débil y dificil de demostrar.

- Strengthening: Sacar hipótesis hace que la afirmación sea más fuerte y fácil de demostrar.

Veamos un ejemplo de Strengthening: $x : Bool \vdash True : Bool$
Hagámosnos una pregunta ¿utilizamos en algún lado la x? No, entonces tenemos que preguntarnos ¿realmente la necesitamos en el contexto? No. Entonces podemos quitarla y la afirmación será aún más fuerte.
Entonces $\emptyset \vdash True : Bool$

0.6. Tipos Habitados

Un tipo está habitado si existe una función de ejemplo que use ese tipo.
Por ejemplo, si nos piden algo que cumpla $\vdash M : (\tau \to \rho) \to (\sigma \to \tau) \to (\sigma \to \rho)$.
Podemos notar que una función que haga esto, sería la composición: $\lambda f : \tau \to e.\lambda g : \sigma \to \tau.\lambda x : \sigma.f(gx)$ o mejor escrita $(\lambda f : \tau \to e)(\lambda g : \sigma \to \tau)(\lambda x : \sigma)(f(gx))$

0.7. Semántica Formal

El sistema de tipos nos indica cómo se construyen los programos pero además queremos darles **significado** (semántica). Existen tres formas de dar semántica formal

- **Semántica Operacional**: Indica cómo se ejecuta el programa hasta llegar a un resultado.
  - Es la que vamos a usar en la materia.
  - Trabaja en base a reglas de reducción $\to$ entre términos. Cuando tenemos $M \to N$, M reduce.
  - La idea es llegar a formas normales, es decir, expresiones irreducibles que no existe ninguna regla que las pueda deducir.
  - small-step: ejecución paso a paso.
  - big-step: evaluación directa al resultado.
- **Semántica Denotacional**: Interpreta los programas como objetos matemáticos.
- **Semántica Axiomática**: Establece relaciones lógicas entre el estado del programa antes y después de la ejecución.

Importante: Nosotros vamos a usar la estrategia de reducción call by value, y nos va a interesar mantener el determinismo de las reglas de reducción. **IMPORTANTÍSIMO**: Cada término **debe reducirse** a un paso a la vez y con **una sola elección** de regla semántica. Si existe más de una regla semántica para un paso dado, no es determinista.

0.8. Semántica Operacional

0.8.1. Small-Step

La semántica operacional predica sobre **juicios de evaluación**: $M \to N$ donde M y N son programas.

0.8.2. Valores

Los valores son los posibles resultados de evaluar programas: $V ::= true \mid false \mid \lambda x : \tau.M$
Los valores ya tienen todos sus parámetros completamente reducidos, de lo contrario, no serían valores.
Ej.: $V ::= \ \ldots \mid Nil \mid Bin(V,V,V)$ corresponde a los valores de trabajar con un árbol binario.

Cantidad de Reglas de Congruencia

Suele ser una de reducción por cada término, pero es importante recalcar que si un término tiene más de un parámetro (sin variables libres) entonces ese término tiene más de una regla de reducción.
A su vez, hay que hacer hincapié en por qué resalté el **sin variables libres**. Esto es porque los términos que tienen variables libres no se reducen, sino que solamente se sustituyen porque no son términos cerrados.
Lo que hay que meterse en la cabeza es que: si no reducimos completamente todo en las reglas de congruencia es: porque o nos falta algo, o se van a sustituir en las reglas de cómputo. TODO debe finalizar siendo un valor.
Ej.: Sean $M, N, O ::= \ldots \mid []_\sigma \mid M :: N \mid FOLDR\ M\ BASE \to N; REC(h,r) \to O$
¿Qué reglas de Congruencia existen para estos términos? bueno, tenemos aparentemente 3.

- Reducir M de M::N
- Reducir N de M::N
- Reducir M del FOLDR. h NO se reduce porque es un valor (cabeza de la lista) y r tampoco, porque r al ser un caso recursivo tiene variables libres.

¿Cuando reducir y cuando no?

- Si tiene variables libres no se reduce.
- Si es un caso condicional, no se reduce, porque de antemano no sabés en qué casos necesitás reducirlo. Por ejemplo, los CASE, IF o FOLDR no se suelen reducir sus términos (sí las guardas), porque no es necesario gastar cómputo en algo que no se sabe si vamos a usar.

Cantidad de Reglas de Cómputo

Todos los valores posibles tienen una regla de Cómputo, usualmente, cada valor va a estar asociada a las posibles operaciones de las reglas de Congruencia.
Siguiendo el ej. anterior, defina los posibles valores del sistema y dé las reglas de cómputo.
Bueno, los posibles valores con listas hay dos: $V ::= []_\sigma \mid V :: V$
¿Qué operaciones deberíamos poder realizar con estos valores?: FOLDR con lista vacía, y FOLDR sin una lista vacía.

- Si el FOLDR tiene lista vacía entonces devuelve simplemente N.
- Si el FOLDR NO tiene lista vacía, entonces, debo sustituir las variables h y r con la información para el paso recursivo.

0.9. Semántica Operacional

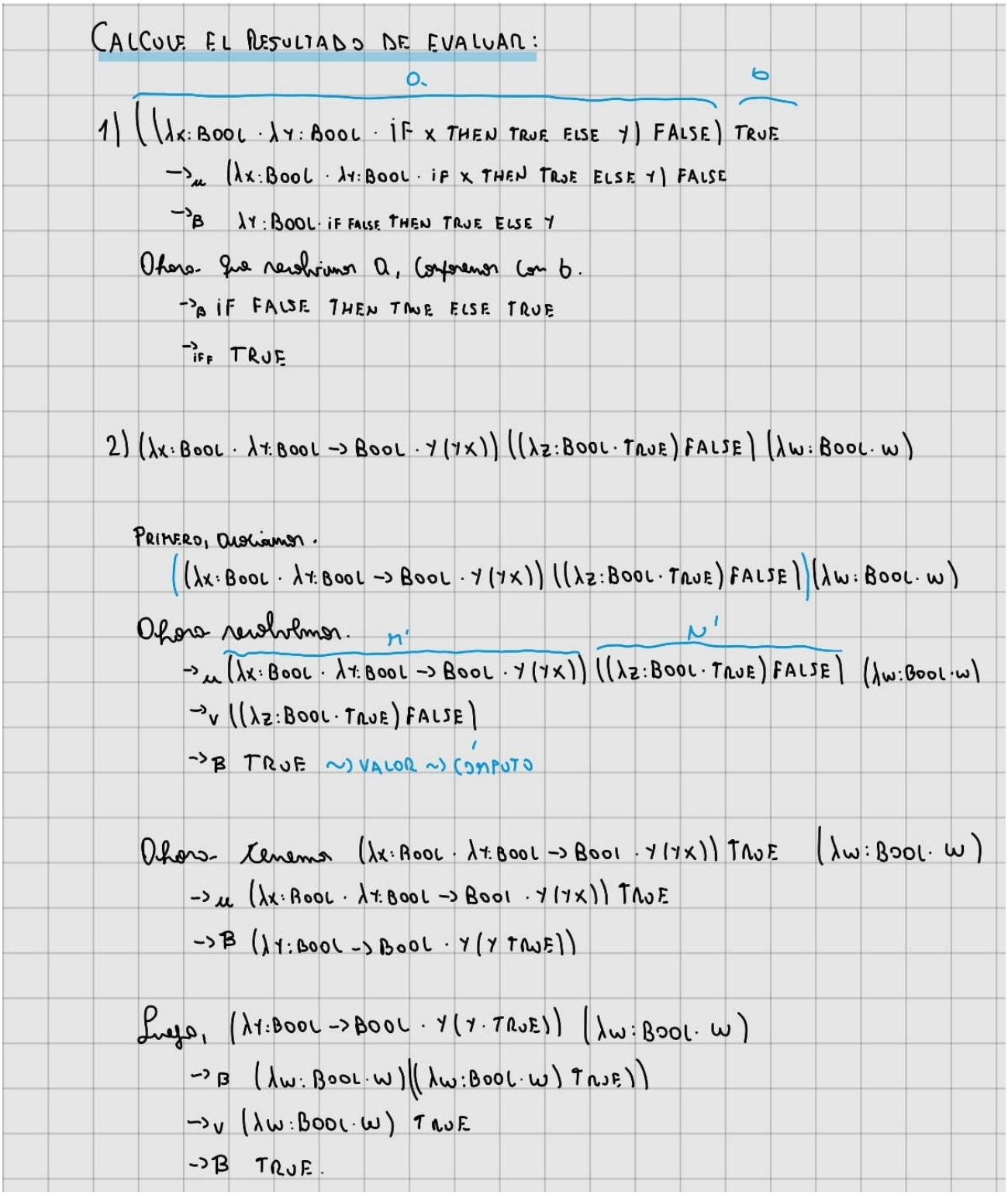
$$(\lambda x : \sigma . M)V \to M\{x := V\} \qquad (\beta)$$
$$\text{if true then } M \text{ else } N \to M \qquad (\text{if}_\text{t})$$
$$\text{if false then } M \text{ else } N \to N \qquad (\text{if}_\text{f})$$

Si $M \to N$, entonces:

$$MO \to NO \qquad (\mu)$$
$$VM \to VN \qquad (\nu)$$
$$\text{if } M \text{ then } O \text{ else } P \to \text{if } N \text{ then } O \text{ else } P \qquad (\text{if}_c)$$

Importante: μ, v, if_c son reglas de congruencia, es decir, nos permiten "ingresar a la expresión para luego aplicar las reglas de cómputo" las cuales, las reglas de compúto son $\beta, (if_t), (if_f)$
Importante 2: Aplicar la regla β NO cuenta como regla de redudcción.
Importante 3: β se utiliza **SOLO** con valores.
Importante 4: Recordar que $\to$ indica a qué reduce, por ejemplo, en el caso del if false, tiene sentido que si el else es N, la operación te devuelva N.

Veamos a continuación unos ejemplos



Reglas de Tipado vs Reglas de Semántica

1. En las Reglas de Tipado **existe** el contexto mientras que en las Reglas de Semántica **no existe**.
La excepción de usar un contexto en las reglas de semántica se da cuando tenemos alguna especie de guarda o un caso donde tenemos que devolver cosas de algún tipo específico, pero sin contexto no tenemos esa información.

$$\frac{\emptyset \vdash V_1 : \sigma}{FROM\ V_1\ UNTIL_x N\ BY\ V_2 \to IF\ N\{x := V_1\}\ THEN\ []_\sigma\ ELSE\ V_1 :: (FROM\ V_2 V_1\ UNTIL_x\ N\ BY\ V_2)}$$

En este ejemplo podemos ver que como estamos en un lenguaje con **tipado estricto**, necesitamos que tanto el THEN como el ELSE devuelvan el mismo tipo. Entonces no podemos devolver una lista vacía sin especificar el tipo de sus elementos.
2. En las Reglas de Semántica no se coloca el tipo de retorno en cada una de las reglas.

0.9.1. Sustitución

La operación de sustitución $M\{x := N\}$ nos devuelve un nuevo término tal que resulta de reemplazar todas las ocurrencias libres de x en M por N.
Nota: Se implementa por pattern matching porque es un constructor recursivo.

$$
\begin{aligned}
x\{x := N\} &\stackrel{\text{def}}{=} N \\
a\{x := N\} &\stackrel{\text{def}}{=} a \text{ si } a \in \{\text{true}, \text{false}\} \cup \mathcal{X} \setminus \{x\} \\
(\text{if } M \text{ then } P \text{ else } Q)\{x := N\} &\stackrel{\text{def}}{=} \text{if } M\{x := N\} \\
&\quad \text{then } P\{x := N\} \\
&\quad \text{else } Q\{x := N\} \\
(M_1\, M_2)\{x := N\} &\stackrel{\text{def}}{=} M_1\{x := N\}\, M_2\{x := N\} \\
(\lambda y : \tau.\, M)\{x := N\} &\stackrel{\text{def}}{=} \\
&\begin{cases}
\lambda y : \tau.\, M & \text{si } x = y \\
\lambda y : \tau.\, M\{x := N\} & \text{si } x \neq y,\ y \notin \text{fv}(N) \\
\lambda z : \tau.\, M\{y := z\}\{x := N\} & \text{si } x \neq y,\ y \in \text{fv}(N), \\
& \quad z \notin \{x, y\} \cup \text{fv}(M) \cup \text{fv}(N)
\end{cases}
\end{aligned}
$$

3. No existen los axiomas como tal en las Reglas de Semántica. Los valores siempre están ligados ante una operación dada que los utiliza.

0.10. Propiedades de la Evaluación

0.10.1. Teorema (Determinismo)

Si $M \to N_1$ y $M \to N_2$ entonces $N_1 = N_2$
En criollo: No hay más de una forma de reducir algo.

0.10.2. Teorema (Preservación de Tipos o Subject Reduction)

Si $\Gamma \vdash M : \tau$ y $M \to N$ entonces $\Gamma \vdash N : \tau$
Si deducimos el tipo τ para un término sin ejecutar el programa, y luego ejecutamos el programa obteniendo N, entonces el término N tiene el mismo tipo.

0.10.3. Teorema (Progreso)

Si $\vdash M : \tau$ entonces:

- O bien M es un término cerrado irreducible bien tipado, osea un valor.
- O bien existe un N tal que $M \to N$

0.10.4. Teorema (Terminación)

Si $\vdash M : \tau$ entonces no hay una cadena infinita de pasos: $M \to M_1 \to M_2 \to ...$

0.10.5. Canonicidad

- Si $\vdash M : bool$ es derivable, entonces la evaluación de M termina y el resultado es true o false.
- Si $\vdash M : \tau \to \sigma$ es derivable, entonces la evaluación de M termina y el resultado es una abstracción.

0.10.6. Extensión con Números Naturales

Sintaxis:

Se extienden las gramáticas de términos y tipos de la siguiente manera:

$$\sigma ::= \ldots \mid \mathsf{Nat}$$
$$M ::= \ldots \mid \mathsf{zero} \mid \mathsf{succ}(M) \mid \mathsf{pred}(M) \mid \mathsf{isZero}(M)$$

Se extiende el sistema de tipado con las siguientes reglas:

$$\frac{}{\Gamma \vdash \mathsf{zero} : \mathsf{Nat}}\ ax_0 \qquad \frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash \mathsf{succ}(M) : \mathsf{Nat}}\ succ$$

$$\frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash \mathsf{pred}(M) : \mathsf{Nat}}\ pred \qquad \frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash \mathsf{isZero}(M) : \mathsf{Bool}}\ isZero$$

Semántica:

Se extienden los valores de la siguiente manera:

$$V ::= \ldots \mid \mathsf{zero} \mid \mathsf{succ}(V)$$

Además, usamos la notación $\underline{n}$ para $\mathsf{succ}^n(\mathsf{zero})$ con $n \geq 0$.
Se extiende la semántica operacional con las siguientes reglas:

$$\mathsf{pred}(\mathsf{succ}(V)) \to V \qquad (\mathsf{pred})$$
$$\mathsf{isZero}(\mathsf{zero}) \to \mathsf{true} \qquad (\mathsf{isZero}_0)$$
$$\mathsf{isZero}(\mathsf{succ}(V)) \to \mathsf{false} \qquad (\mathsf{isZero}_n)$$

Si $M \to N$, entonces:

$$\mathsf{succ}(M) \to \mathsf{succ}(N) \qquad (\mathsf{succ}_c)$$
$$\mathsf{pred}(M) \to \mathsf{pred}(N) \qquad (\mathsf{pred}_c)$$
$$\mathsf{isZero}(M) \to \mathsf{isZero}(N) \qquad (\mathsf{isZero}_c)$$

Ultra Importante: Cada vez que extendmos tipos o sintaxis hay que chequear que siga valiendo el progreso, el determinismo y la preservación de tipos.
¿Es segura esta extensión?: No, porque con estas reglas se rompe el progreso pues pred(0) es forma normal pero no es un valor válido.
Si quisieramos que siga valiendo el progreso podemos extender la semántica y agregar un caso $pred(zero) \to zero$ **(**$pred_0$**)**

0.10.7. Macros

Podemos definir macros para expresiones que vayamos a utilizar con frecuencia. Por ejemplo:

- ✨ $Id_\tau \stackrel{def}{=} \lambda x : \tau . x$
- ✨ and $\stackrel{def}{=} \lambda x : \mathsf{Bool} . \lambda y : \mathsf{Bool} . \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ \mathsf{false}$

0.10.8. Agregando Reglas para las Abstracciones

Cambiando reglas semánticas

Supongamos que agregamos la siguiente regla para las abstracciones:
Si $M \to N$, entonces:

$$\lambda x : \tau . M \to \lambda x : \tau . M' \qquad (\zeta)$$

Ejercicio

1. Repensar el conjunto de valores para respetar esta modificación, pensar por ejemplo si λx: Bool.Id_{Bool} true es o no un valor. ¿Y λx: Bool.x?

   $V ::= \text{true} \mid \text{false} \mid \lambda x : \sigma . F \mid 0 \mid \text{succ}(V)$, donde F es una forma normal.

2. ¿Qué reglas deberían modificarse para no perder el determinismo?

   $$(\lambda x : \sigma . F)V \to F\{x := V\} \qquad (\beta)$$

3. Utilizando la nueva regla y los valores definidos, reducir la expresión:
   λz: Nat $\to$ Nat.$(\lambda x$: Nat $\to$ Nat.$z\ \underline{23})\ \lambda x$: Nat.$0$

(No entendí qué caso es el conflictivo si tuvimos que cambiar de $\lambda x : \sigma . M$ a $\lambda x : \sigma . F$) ¿qué no podíamos hacer antes?
Entiendo que diciendo que tengas $\lambda x : \sigma . F$ estas diciendo que si le metés un input tenés que garantizar que lo que tenga la lambda adentro **ya sea una forma normal** y sabés que el proceso va a terminar bien.
Si usás $\lambda x : \sigma . M$ tenés el problema de que si le ponés cualquier cosa a M podés llegar a algo que no es reducible y capaz se cuelga?

Sistema de Tipos para el Cálculo Lambda

Aclaración Importante: En deducción natural hablabamos de τ, σ como hipótesis o proposiciones. Acá en cálculo lambda, esos son tipos.
Ej.: $\tau \to \sigma$ era una conclusión en deducción natural, acá es el tipo de una lambda.
Entonces ahora sí, veamos el sistema de tipos para el cálculo Lambda omitiendo los booleanos porque son exactamente igual que en deducción natural.

$$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x : \tau . M : \tau \to \sigma}\ \text{T-ABS } (\to i) \qquad \frac{}{\Gamma, x : \tau \vdash x : \tau}\ \text{T-VAR (ax)} \qquad \frac{\Gamma \vdash M : \tau \to \sigma r \vdash N : \tau}{\Gamma \vdash MN : \tau}\ \text{T-APP } (\to \text{e})$$

0.10.9. Lógica Combinatoria

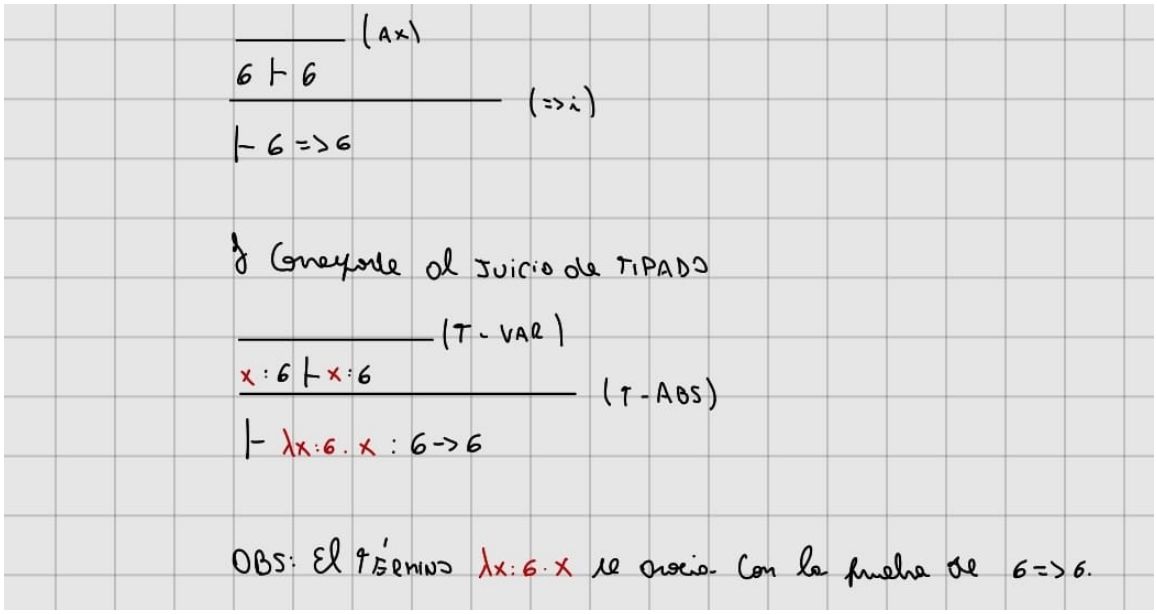
Es una variante del calculo λ.
Si a $\sigma \to \tau$ lo leemos como $\sigma \implies \tau$ luego, la regla de tipadod e aplicación de una función es la regla Modus Ponens.

Pruebas y Programas

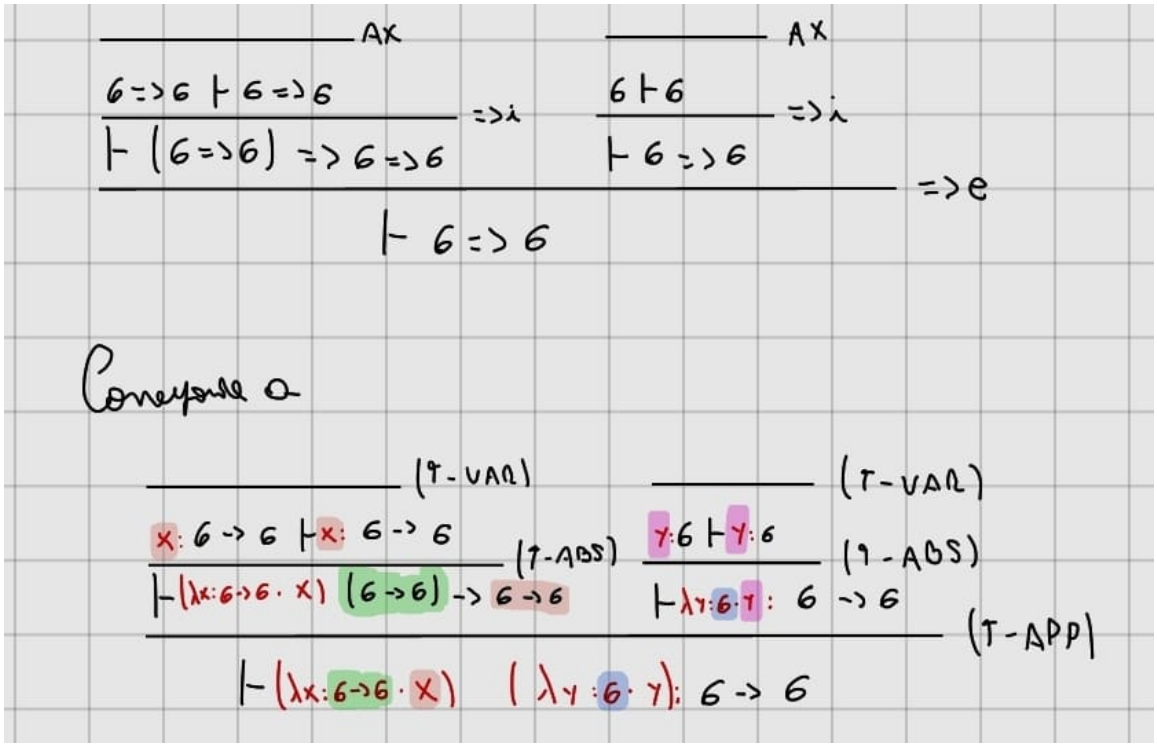
En la **Deducción Natural** hablamos en base a proposiciones, en cálculo lambda, las proposiciones son tipos.
En la **Deducción Natural** hablamos en base a pruebas, en cálculo lambda, las pruebas son términos.

0.10.10. Juicios Derivables

Un juicio $\vdash \tau$ es derivable sí y solo sí, el tipo τ está habitado, es decir, existe algun camino que nos lleva a demostrar τ.
Como también podemos probar estas cosas en deducción natural, lo hacemos también en cálculo lambda.



Importante: Como esto son fórmulas, puede haber muchas pruebas distintas para una misma fórmula. Mirá esta forma de probar lo mismo de antes pero más complicado.



0.10.11. Simplificación de Pruebas

Corte: Un corte es un lema que probamos a pesar que **no es una subfórmula de la afirmación final**. Es decir, no es algo que queramos demostrar en la conclusión pero nos ayuda a seguir adelante.
Se caracteriza por tener pasos como $\Longrightarrow_i$ y luego $\Longrightarrow_e$ o también, $\wedge_i$ y $\wedge e_1$ o $\wedge e_2$.
Básicamente cuando veas una introducción y una eliminación, ahí estás haciendo un corte.
La variable que hacés aparecer aparte es el corte.
En la computación, un paso de reducción β (E-APP ABS) se corresponde con una eliminación de corte.

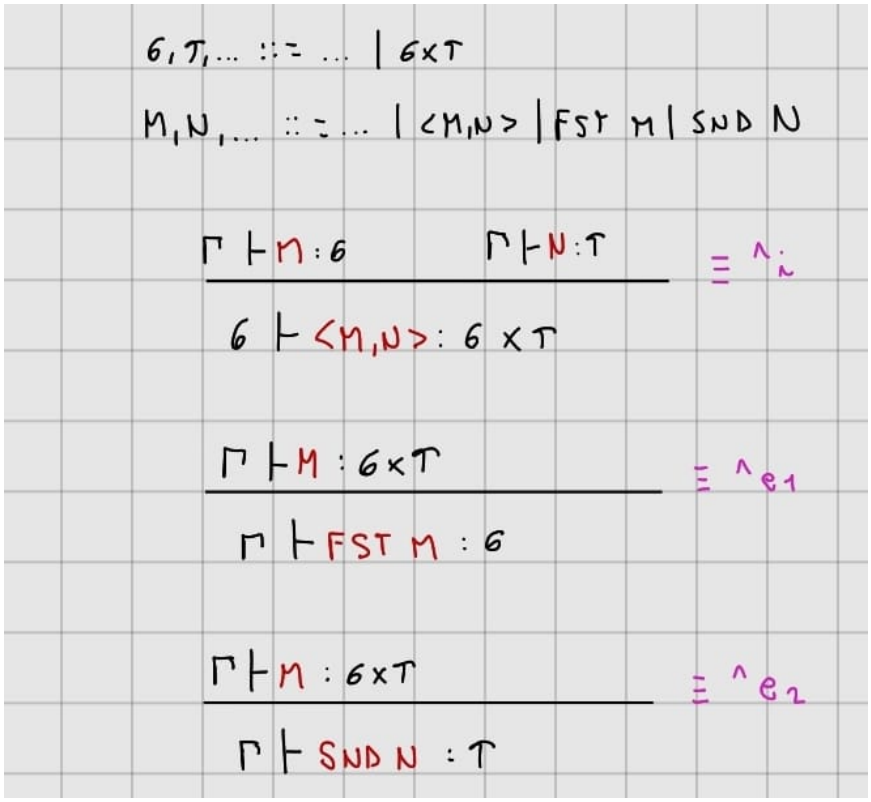
0.10.12. Prueba en Forma Normal

Es aquella prueba que no posee ningún tipo de corte.

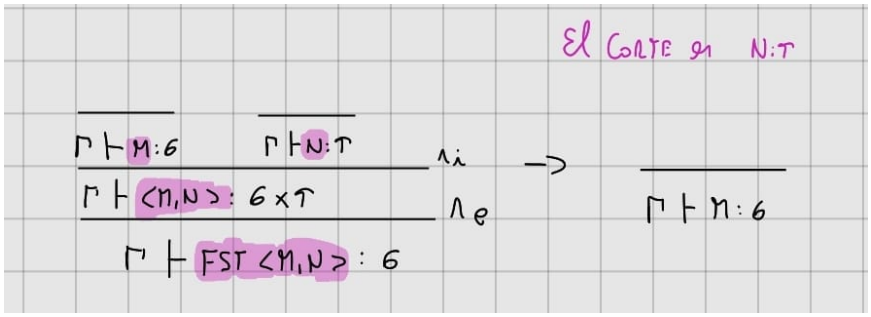
0.10.13. Normalización de Pruebas

Se realiza mediante la eliminación de cortes, es decir, tratar de quitar esos cortes para llevarlo a forma normal.

0.10.14. Conjunción (Producto) en Lambda



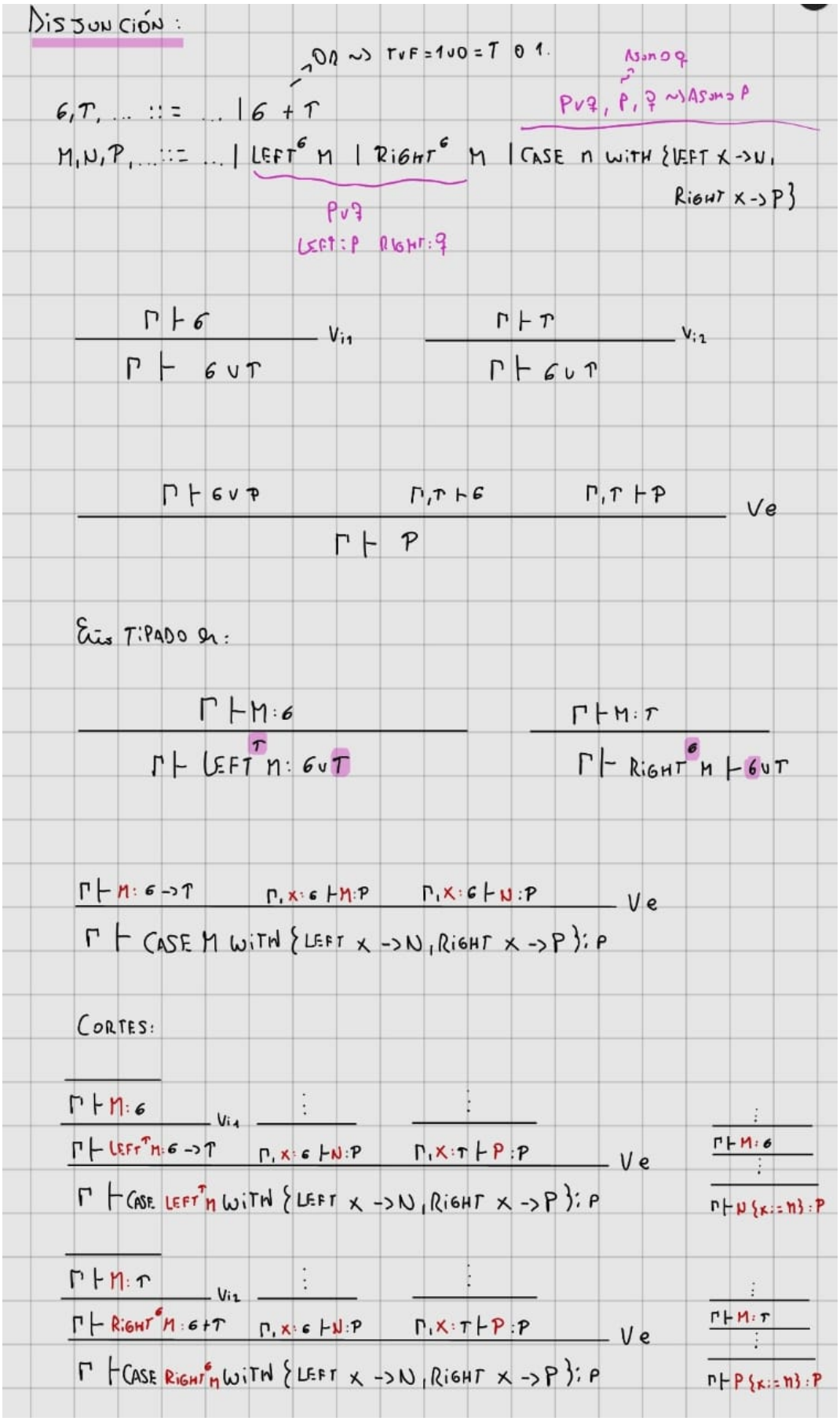
Veamos ahora como hacer reducciones con productos.



Nótese que la idea desde un primer momento era quedarse con M σ. Por lo tanto, el corte es añadir τ.

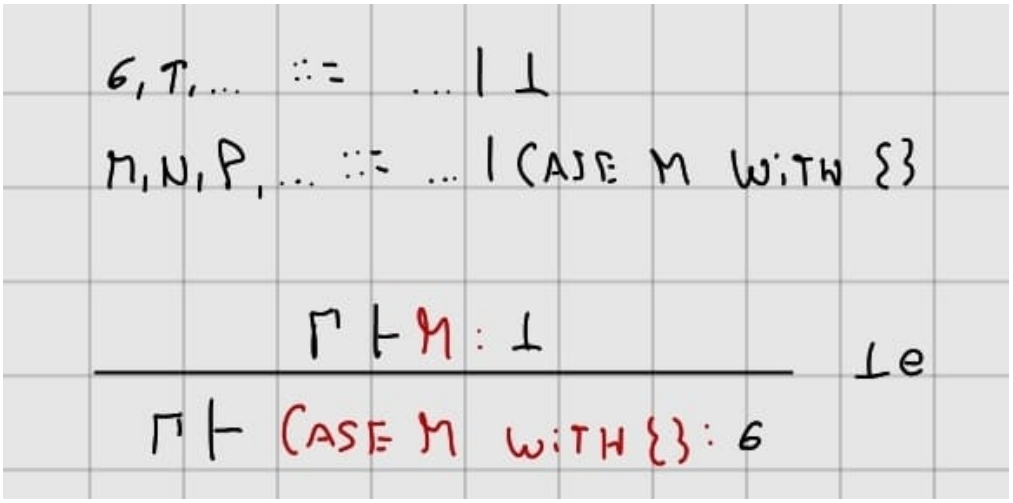
0.10.15. Disyunción (Suma) en Lambda

El corte característico acá es $\vee_e$ y luego $\vee_i$ mirando desde abajo hacia arriba.



0.10.16. Absurdo

No existe **ningún constructor** para el tipo $\bot$ (vacío).



Correspondencia de Curry-Howard

$A_1, ...A_n \vdash \sigma$ es derivable en NJ (Logica Intusionista) si $\exists$ término M donde $Fv(M) \subseteq \{x_1, ..., x_n\}$ tal que $x_1 : A_1, ..., x_n : A_n \vdash M : \sigma$
En criollo: Si en Cálculo Lambda tenes términos $x_1, ..., x_n$ que correspondan a $A_1, ..., A_n$ uno a uno entonces, lo podés considerar como válido.
Ej.: $P, Q \vdash P \wedge Q \equiv x : P, y : Q \vdash < x, y >: P \wedge Q$ donde

- $A_1 = P$ $A_2 = Q$
- $x_1 : P$ es decir, $x_1 : A_1$
- $x_2 : Q$ es decir, $x_2 : A_2$
- $\sigma = P \wedge Q$ pero en cálculo lambda, nuestro término M, es de tipo sigma. $< x, y >: P \wedge Q$

0.10.17. Corolario de Logica Intusionista

$\nvdash \bot$: No puedo concluir que hay contradicción, lo que sugiere que mis premisas son consistentes.

0.10.18. Codificación de la Negación

$\neg\sigma \equiv \sigma \to \bot$
Nota: $\neg_i \equiv \Longrightarrow\ i$ y $\neg_e \equiv \Longrightarrow\ e$

0.10.19. Tipo Unit

También conocida como Fórmula T, es una fórmula válida que siempre es verdadera y es un tipo algebraico con un único constructor T.
En el cálculo λ extendemos la sintaxis con el tipo T que tiene un único elemento.

$$\begin{array}{lll} \sigma, \tau, \ldots & ::= & \ldots \mid \top \\ M, N, P, \ldots & ::= & \ldots \mid \top \end{array}$$

Consideramos a la Logica Intusionista extendida con la siguiente regla

$$\frac{}{\Gamma \vdash T}\ T_i$$

Y tiene la siguiente regla de tipado en Cálculo λ

$$\frac{}{\Gamma \vdash T : T}\ \text{T-UNIT}$$

0.10.20. Recursión

Se utiliza el operador fix: $M ::= ... \mid fix\ M$
No se precisa ningún tipo nuevo pero sí una regla de tipado.

$$\frac{\Gamma \vdash M : \sigma \to \sigma}{\Gamma \vdash fix\ M : \sigma}\ \text{T-FIX}$$

También extendemos la semántica para poder ir resolviendo las operaciones.

$$\frac{M \to M'}{\text{fix } M \to \text{fix } M'}\ \text{E-FIX}$$

$$\frac{}{\text{fix } (\lambda x : \sigma.M) \to M\{x := \text{fix } (\lambda x : \sigma.M)\}}\ \text{E-FIXBETA}$$

Gracias a fix ahora podemos definir funciones parciales $fix\ (\lambda x : \sigma.x)$ donde $\vdash\ fix\ (\lambda x : \sigma.x) : \sigma$ para cualquier σ. En particular, vale para $\sigma = \bot$
Importante: No podemos extender la Lógica Intusionista (NJ) con un operador fix pues la lógica sería inconsistente. Ej.: $\vdash \bot$ sería derivable.

Inferencia de Tipos

Términos sin anotaciones de tipos: U :: = x — $\lambda x.U$ — U U — True — False — if U then U else U
Términos con anotaciones de tipos: U :: = x — $\lambda x.M$ — M M — True — False — if U then U else U

erase(M)

Es el término sin anotaciones de tipos que resulta de borrarlos del término M. El resultado es U.
Ej.: $erase((\lambda x : Bool.x)\ True) = ((\lambda x.x)\ True)$
Esto nos quiere decir que, si tenemos un término M es fácil conseguir U pero no al revés.

¿Cuándo podemos tipar un término U?

Un término U sin anotaciones de tipos es tipable sí y solo sí existen:

- Un contexto de tipado Γ
- Un término con anotaciones de tipos M
- Un tipo τ tales que erase(M) = U y $\Gamma \vdash M : \tau$

Entonces, el problema de **inferencia de tipos** consiste en:

- Dado un término U determinar si es tipable.
- En caso de que U sea tipable, entonces
  - Hallar un contexto Γ, un término M y un tipo τ tales que cuando hagamos erase(M) = U y $\Gamma \vdash M : \tau$

En criollo: Si tengo un U, entonces buscate un M, que tenga contexto, un τ y que si le mando ese M me da U.
Ej.: $\lambda x.x$ es la función identidad. ¿Cómo podemos tiparla?. De antemano, x podría ser cualquier cosa, una función, un valor, lo que realmente quiera.
Entonces para poder tiparla, necesitamos de antemano, saber en el contexto de qué tipo es x, por lo tanto, podemos decir que $\Gamma \vdash \{x : \alpha\}$.
Ya tenemos entonces Γ y U. Nos falta conseguir τ que sería lo que nos falta para poder concluir $\Gamma \vdash M : \tau$. Como es una Lambda que recibe un x, y devuelve un x, entonces el tipo es $\tau : \alpha \to \alpha$.
Entonces, armamos el término M: $\lambda(x : \alpha).x$ y podemos chequear ambas condiciones.

- erase($\lambda(x : \alpha).x$) = $\lambda x.x$ y esto es verdadero.
- $\Gamma \vdash (\lambda(x : \alpha).x) : \alpha \to \alpha$

$$\frac{\Gamma, x : \alpha \vdash x : \alpha}{\Gamma \vdash (\lambda(x : \alpha).x) : \alpha \to \alpha}\ \text{T-ABS}$$

y es justamente lo que queríamos derivar.

Importante: Solo tipo en M si $\exists$ una lambda. Ej.: $U = x\ True \equiv M = x\ True$ pues no existe ninguna lambda en U.

Incorporando incógnitas a los tipos

El algoritmo de inferencia de tipos se basa en manipular tipos **parcialmente conocidos**.
Ej.: $x\ True$, ¿cómo podemos tiparlo?. Bueno, recordemos que asociamos a la izquierda, por lo tanto sería algo así: $x(True)$ por lo tanto, x debería de ser una función que recibe obligatoriamente un booleano y devuelve un tipo desconocido. Ese tipo desconocido lo denotamos con una variable.
Por lo tanto, el tipo sería: (x: $Bool \to X_1$)
Ej. 2: $(X_1 \to X_1) \stackrel{?}{=} ((Bool \to Bool) \to X_2)$.
En criollo lo que nos pide es: ¿cómo tiene que ser X1 y X2 para que ambos lados se verifiquen? bueno, si vemos la ecuación izquierda antes de $\to$ tiene $Bool \to Bool$ entonces no queda otra que $X_1 = Bool \to Bool$ Así mismo, entonces, del lado izquierdo nos queda $Bool \to Bool \to Bool \to Bool = (Bool \to Bool)X_2$ entonces la única forma que tiene X_2 es $Bool \to Bool$.
Por lo tanto, $X_1 : Bool \to Bool$ y $X_2 : Bool \to Bool$
Ej. 3: if xy then y else true
La estrategia en los condicionales, o lugares donde hay ramas es la siguiente: **considerar y tipar cada rama por separado, luego se hace puesta en común**.
Entonces, en este caso tenemos que tipar tres ramas, una incógnita por cada variable:

- xy: $\Gamma_c = \{x : X_1 \to X_2, y : X_1\}$
- y: $\Gamma_t = \{t : X_3\}$
- true: $\Gamma_e = \emptyset$

Entonces ahora planteamos un sistema de ecuaciones $\begin{cases} X_3 \stackrel{?}{=} Bool \\ X_1 \stackrel{?}{=} X_3 \\ X_2 \stackrel{?}{=} Bool \end{cases}$
Sustituimos ahora las incógnitas por los tipos y nos queda $X_1 = Bool, X_2 = Bool, X_3 = Bool$.
Resolver ecuaciones del tipo $X_1 \stackrel{?}{=} X_2 \to X_3$ se llama UNIFICACIÓN. Reemplazando las variables deberíamos llegar a una igualdad.

Unificación

Es el problema de resolver sistemas de ecuaciones entre tipos con incógnitas. Es muy importante pues luego lo usamos para dar un algoritmo de inferencia de tipos.
Tenemos conjuntos finitos de constructores de tipos

- Tipos constantes (no tienen parámetros): Bool, Int, ...
- Constructores Unarios (reciben un parámetro): (List •), (Maybe •), ...
- Constructores Binarios (reciben dos parámetros): $(\bullet \to \bullet)$, $(\bullet \times \bullet)$, $(Either\ \bullet\ \bullet)$

Los **tipos** se forman usando incógnitas y constructores: $\tau ::= X_n \mid C(\tau_1, \ldots, \tau_n)$
Antes de definir formalmente la Unificación, primero veamos la Sustitución, que es una de las herramientas que vamos a usar.

Sustitución

Es una función que a cada incógnita le asocia un tipo.
Notamos: $\{T_1 \stackrel{?}{=} \sigma_1, \ldots, T_k \stackrel{?}{=} \sigma_k\}$ a la sustitución S tal que $S(T_1) = S(\sigma_1) \wedge \cdots \wedge S(T_k) = S(\sigma_k)$
Si σ es un tipo, escribimos $S(\sigma)$ para que cada incógnita de σ sea reemplazada por lo que haya sido definido en S.

Ej.: Si $S = \{X_1 := Bool, X_3 := (X_2 \to X_2)\}$ entonces S$((X_1 \to Bool) \to X_3) = (Bool \to Bool) \to (X_2 \to X_2)$

Unificación

Un problema de unificación es un conjunto finito E de ecuaciones entre tipos que pueden involucrar incógnitas.
Un **unificador** para E es una sustitución **S** tal que

- $S(\tau_1) = S(\sigma_1)$
- ...

- $S(\tau_n) = S(\sigma_n)$

Ej.: $\{X_1 \stackrel{?}{=} X_2\}$ mientras que sea una igualdad, se puede decir cualquier cosa de X_1 y X_2. Todas serán solución pero si la solución tiene menos variables para solucionar el problema, es más general.

Sustitución más general

Una sustitución S_A es **más general** que una sustitución S_B si existe una sustitución S_C tal que: $S_B = S_C \circ S_A$
En Criollo: Si S_B nace de S_A particularmente, entonces S_A es más general.

Algoritmo de unificación Martelli-Montanari (M-M) y la Corrección del Algoritmo

Consiste en aplicar secuencialmente un conjunto de reglas las cuales hay dos chances: o falla, o sigue adelante reemplazando. Este algoritmo siempre termina para cualquier problema de unificación E.

- Mientras que $E \neq \emptyset$ se aplica sucesivamente alguna de las reglas.
- La regla puede resultar en una falla (si E no tiene solución)
- De lo contrario, la regla es de la forma $E \to_s E'$ y al aplicarlo sucesivamente, si no hay falla, el algoritmo llega a $\emptyset$:
  - $E = E_0 \to_{S_1} E_1 \to_{S_2} E_2 \cdots \to_{S_n} E_n = \emptyset$ $S = S_n \circ S_{n-1} \cdots \circ S_2 \circ S_1$ es un unificador para E, es el más general posible y lo denotamos mgu(E).

Reglas del Algoritmo (M-M)

$$\{X_n \stackrel{?}{=} X_n\} \cup E \xrightarrow{\texttt{Delete}} E$$

$$\{C(\tau_1, \ldots, \tau_n) \stackrel{?}{=} C(\sigma_1, \ldots, \sigma_n)\} \cup E \xrightarrow{\texttt{Decompose}} \{\tau_1 \stackrel{?}{=} \sigma_1, \ldots, \tau_n \stackrel{?}{=} \sigma_n\} \cup E$$

$$\{\tau \stackrel{?}{=} X_n\} \cup E \xrightarrow{\texttt{Swap}} \{X_n \stackrel{?}{=} \tau\} \cup E \quad \text{si } \tau \text{ no es una incógnita}$$

$$\{X_n \stackrel{?}{=} \tau\} \cup E \xrightarrow[\{X_n := \tau\}]{\texttt{Elim}} E' = \{X_n := \tau\}(E) \quad \text{si } X_n \text{ no ocurre en } \tau$$

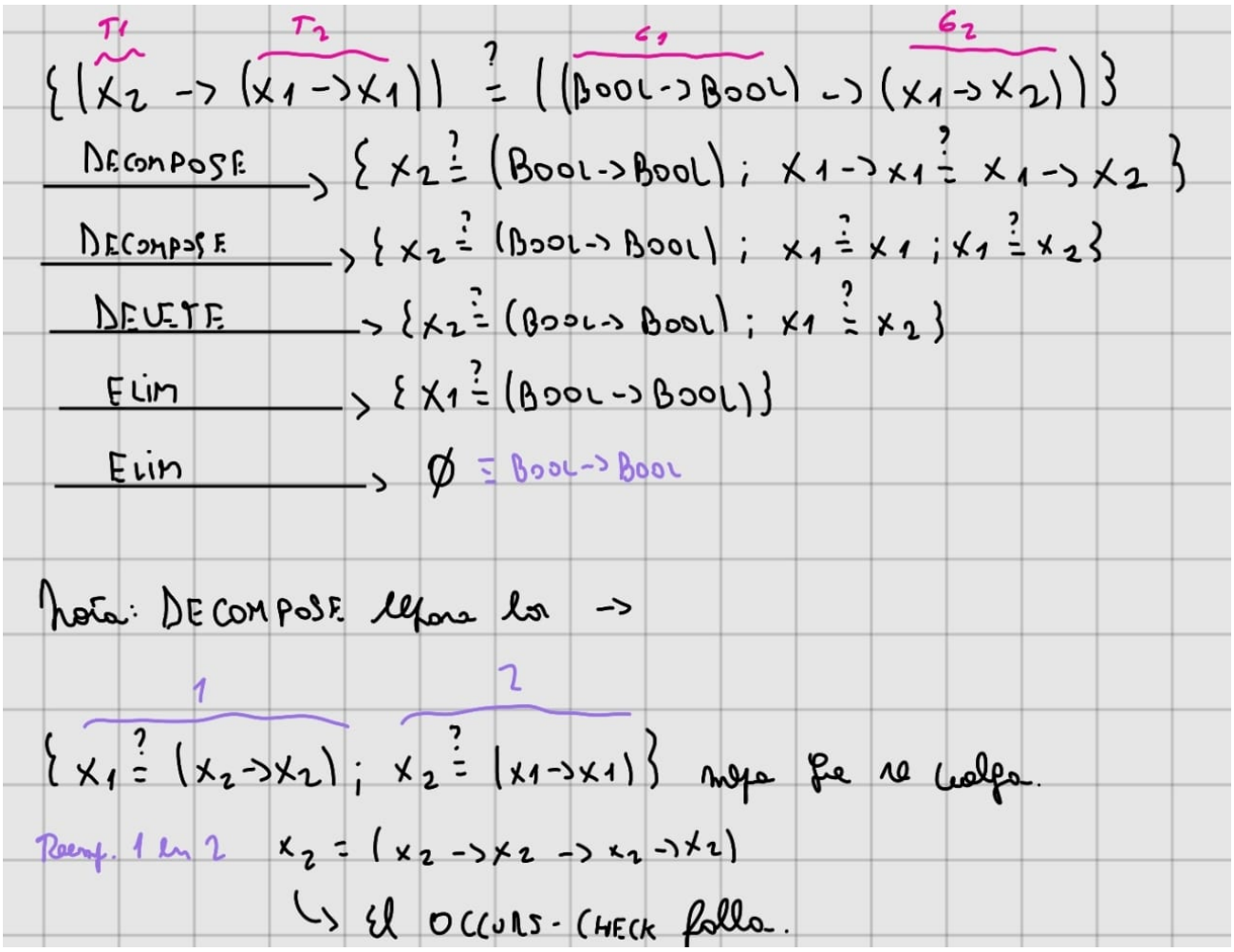
$$\{C(\tau_1, \ldots, \tau_n) \stackrel{?}{=} C'(\sigma_1, \ldots, \sigma_m)\} \cup E \xrightarrow{\texttt{Clash}} \texttt{falla} \quad \text{si } C \neq C'$$

$$\{X_n \stackrel{?}{=} \tau\} \cup E \xrightarrow{\texttt{Occurs-Check}} \texttt{falla} \quad \text{si } X_n \neq \tau \text{ y } X_n \text{ ocurre en } \tau$$

¿Por qué los condicionales en las reglas?

- Swap: τ no debe ser una variable. Debe ser un valor porque sino entramos en un loop infinito de hacer referencia a sí mismo. Ej.: $4 = x \equiv x = 4$ pero x = y ≠ y = x NO es lo mismo porque NO son valores.
- Elim: No podemos reemplazar un término que depende de sí mismo y reducirlo. Ej.: $X_1 = X_1 \to Bool$ no tiene sentido sustituir en este caso porque vamos a estar en prácticamente un loop infinito.
- Clash: Si no tienen la misma cantidad de elementos falla.
- Occurs-Check: Solo podemos unificar si la incógnita no aparece en τ. Ej.: $X_1 = X_1 \to Bool$ falla pero $X_1 = Int$ funciona.

La regla Decompose es muy útil para separar funciones. La Delete la hacemos cuando nos queda una igualdad trivial. La Elim la usamos dejando todo menos la igualdad que queremos quitar.



Nota: El MGU del primer ejercicio sería lo mínimo indispensable para cumplir la igualdad. En este caso sería $X_1 : Bool \to Bool,\ X_2 : Bool \to Bool$
Importante. El algoritmo de simplificación **no siempre** devuelve el mismo resultado si existe una solución.

Algoritmo W (Inferencia de Tipos)

La idea es siempre empezar haciendo las reglas de tipado. La regla de tipado deriva el algoritmo de inferencia.
Al igual que el Algoritmo M-M hay dos opciones

- Puede fallar si U no es tipable.
- Puede tener éxito y si lo tiene devuelve una tripla (Γ, M, τ) donde $erase(M) = U$ y $\Gamma \vdash M : \tau$ ¡sí! es lo mismo que vimos al principio sin ningún algoritmo

Se denota $W(U) \rightsquigarrow \Gamma \vdash M : \tau$ para indicar que el algoritmo de inferencia tiene éxito cuando se le pasa U como entrada y devuelve la tripla.

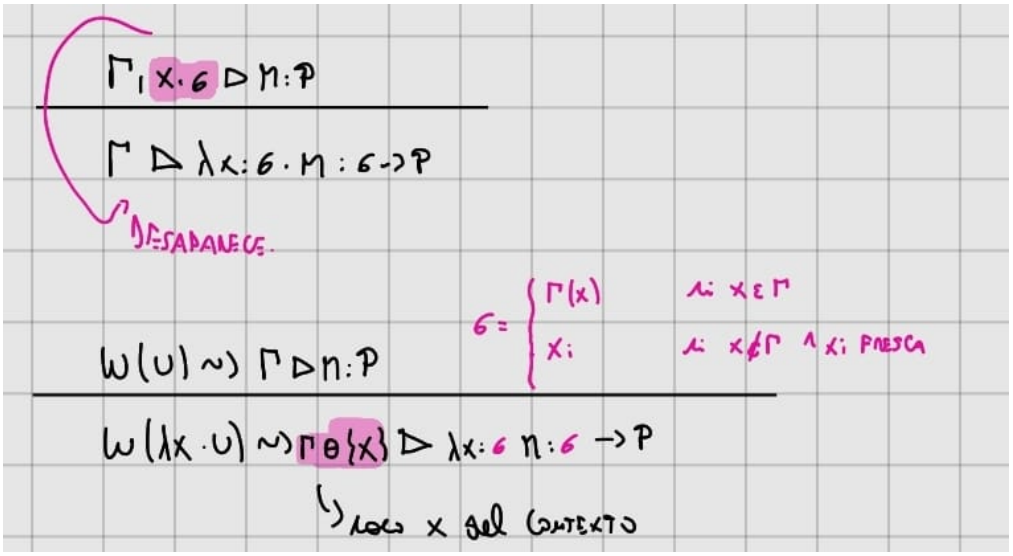
- $\mathbb{W}(x) \rightsquigarrow \{x : \mathrm{X}_k\} \vdash x : \mathrm{X}_k, \quad \mathrm{X}_k$ incógnita fresca
- $\mathbb{W}(0) \rightsquigarrow \emptyset \vdash 0 : Nat$
- $\mathbb{W}(true) \rightsquigarrow \emptyset \vdash true : Bool$
- $\mathbb{W}(false) \rightsquigarrow \emptyset \vdash false : Bool$
- $\mathbb{W}(succ(U)) \rightsquigarrow S(\Gamma) \vdash S(succ(M)) : Nat$ donde
  - $\mathbb{W}(U) = \Gamma \vdash M : \tau$
  - $S = MGU\{\tau \stackrel{?}{=} Nat\}$
- $\mathbb{W}(pred(U)) \rightsquigarrow S(\Gamma) \vdash S(pred(M)) : Nat$ donde
  - $\mathbb{W}(U) = \Gamma \vdash M : \tau$
  - $S = MGU\{\tau \stackrel{?}{=} Nat\}$
- $\mathbb{W}(iszero(U)) \rightsquigarrow S(\Gamma) \vdash S(iszero(M)) : Bool$ donde
  - $\mathbb{W}(U) = \Gamma \vdash M : \tau$
  - $S = MGU\{\tau \stackrel{?}{=} Nat\}$
- $\mathbb{W}(if\ U\ then\ V\ else\ W) \rightsquigarrow S(\Gamma_1) \cup S(\Gamma_2) \cup S(\Gamma_3) \vdash S(if\ M\ then\ P\ else\ Q) : S(\sigma)$ donde
  - $\mathbb{W}(U) = \Gamma_1 \vdash M : \rho$
  - $\mathbb{W}(V) = \Gamma_2 \vdash P : \sigma$
  - $\mathbb{W}(W) = \Gamma_3 \vdash Q : \tau$
  - $S = MGU\{\sigma \stackrel{?}{=} \tau, \rho \stackrel{?}{=} Bool\} \cup \{\sigma_1 \stackrel{?}{=} \sigma_2 \mid x : \sigma_1 \in \Gamma_i, x : \sigma_2 \in \Gamma_j, i, j \in \{1, 2, 3\}\}$
- $\mathbb{W}(\lambda x.U) \rightsquigarrow \Gamma' \vdash \lambda x : \tau'.M : \tau' \to \rho$ donde
  - $\mathbb{W}(U) = \Gamma \vdash M : \rho$
  - $\tau' = \begin{cases} \alpha \text{ si } x : \alpha \in \Gamma \\ \mathrm{X}_k \text{ con } \mathrm{X}_k \text{ variable fresca en otro caso} \end{cases}$
  - $\Gamma' = \Gamma \ominus \{x\}$
- $\mathbb{W}(U\,V) \rightsquigarrow S(\Gamma_1) \cup S(\Gamma_2) \vdash S(M\,N) : S(\mathrm{X}_k)$ donde
  - $\mathbb{W}(U) = \Gamma_1 \vdash M : \tau$
  - $\mathbb{W}(V) = \Gamma_2 \vdash N : \rho$
  - X_k variable fresca
  - $S = MGU\{\tau \stackrel{?}{=} \rho \to \mathrm{X}_k\} \cup \{\sigma_1 \stackrel{?}{=} \sigma_2 \mid x : \sigma_1 \in \Gamma_1, x : \sigma_2 \in \Gamma_2\}$

Realmente Importante: Recordemos que en los condicionales o cosas donde hay varios casos, cada tipo debe ser diferente aunque parezca obvio. Las restricciones se colocan luego. Nótese que en la segunda imagen, U_2 y U_3 son de tipos distintos $W(U_2) = \Gamma_2 \vdash M_2 : \tau_2$ y $W(U_3) = \Gamma_3 \vdash M_3 : \tau_3$ pero la restricción de como son se le da a la hora de hacer la sustitución $\{t_2 = t_3\}$

Realmente Importante: Como en los casos condicionales puede ser que una variable aparezca en más de una rama, tenemos que verificar que esa variable tenga el mismo tipo en todas las ramas. Esto se puede ver en la segunda imagen marcado en rojo.
Imaginemos que en U1, U2, y U3 tengamos apariciones de una misma variable pero con diferente tipo, esto no sería válido.

Realmente Importante: $\emptyset$ en el Algoritmo de W representa quitar una variable específica del contexto. Esto se ve fácil si de

antemano vemos como es el juicio de tipado de la regla.



Incógnitas Frescas

Las Incógnitas Frescas son variables de tipo que se introducen durante el proceso de inferencia para representar tipos que aún no se conocen. Que sea fresca garantiza que es única y no ha sido utilizada antes en el contexto de inferencia actual.

Lógica de Primer Orden (LPO)

Es prácticamente igual a la Lógica Proposicional pero acá se introducen los cuantificadores $\forall$ y $\exists$. En cuanto a tema de materia, es parecido a deduccion natural pero con cuantificadores.
Necesitamos entender la Lógica de Primer Orden porque es en lo que se basa Prolog (Programación Lógica)

Lenguaje de Primer Orden $\mathcal{L}$

Está formado por

- Conjunto de Símbolos de Función $\mathcal{F} = \{f, g, h, \dots\}$. Cada Símbolo de función tiene asociada una aridad (≥ 0)
  - Los Símbolos de Función con aridad 0 se llaman **constantes**
- Conjunto de Símbolos de Predicado $\mathcal{P} = \{P, Q, R, \dots\}$. Cada Símbolo de predicado tiene asociada una aridad (≥ 0)
- Conjunto Infinito de Numerable de Variables $\mathcal{X} = \{X, Y, Z, \dots\}$

Importante: Los predicados solamente definen una relación entre dos elementos de mi dominio (devuelven un valor de verdad). La idea es enviar como argumento cosas que ya son irreducibles o pueden reducirse hasta poder compararse.
Importante: Las funciones solamente reciben como argumentos elementos de mi dominio y devuelven un elemento de mi dominio. Si se quisiese usar para una fórmula lógica ese valor, debería colocarse un predicado que nos arroje un valor de verdad para ese valor del dominio.

Términos de Primer Orden

Definimos el conjunto $\mathcal{T}$ de términos mediante la siguiente gramática $t ::= X \mid f(t_1, \dots, t_n)$ donde **X** denota una variable y **f** un símbolo de función de aridad n.
Importante: Si un símbolo de función, llamémosle g tiene aridad 3, se debe usar enviando los 3 parámetros. Acá no existe la opción de algo ser opcional.
Véase **anexo** para poder ver con más profundidad la justificación de qué es o no un término.

Diferencia entre Símbolo de Función y Símbolo de Predicado

Un Símbolo de Función me devuelve un elemento de mi Dominio o universo. Los predicados no.
Ej.: $n,\ succ(n)$ son símbolos de función mientras que $\geq$ es un símbolo de predicado.

Notación Infija, Prefija en LPO

Ej.: $+(0, succ(X)) \equiv 0 + succ(X)$. Nosotros usaremos la notación infija.

Fórmulas de Primer Orden

$$
\begin{array}{rcll}
\sigma & ::= & \mathbf{P}(t_1, \ldots, t_n) & \textbf{fórmula atómica} \\
& | & \bot & \text{contradicción} \\
& | & \sigma \Rightarrow \sigma & \text{implicación} \\
& | & \sigma \wedge \sigma & \text{conjunción} \\
& | & \sigma \vee \sigma & \text{disyunción} \\
& | & \neg\sigma & \text{negación} \\
& | & \forall X.\, \sigma & \textbf{cuantificación universal} \\
& | & \exists X.\, \sigma & \textbf{cuantificación existencial}
\end{array}
$$

Al igual que en Cálculo Lambda, los cuantificadores ligan variables siempre y cuando esté definida en ese ámbito. En este caso, **X** está ligada; **P** denota un símbolo de predicado de aridad n.
Importante: Dos fórmulas que solo difieren en los nombres de las **variables ligadas** se consideran iguales. Esto es por el isomorfismo de las variables ligadas, da igual qué nombre tengan.

Sustitución

Hay que tener cuidado en sustituir en LPO, si vamos a sustituir una variable que depende de otra que está ligada a un cuantificador, antes hay que renombrar la variable ligada en cuantificador.
Ej.: $\sigma := succ(X) = Y \implies \exists Z.X + Z = Y$
¿Qué es lo que sucede si hacemos $\sigma\{X := Z * Z\}$? Cuando querramos reemplazar en el cuantificador del existe, el existe **ya tiene una Z** pero esta Z que está en el cuantificador es diferente a la que está en nuestro $\{X := Z * Z\}$ por lo tanto hay que renombrar la Z del cuantificador porque de lo contrario, parecerá que son la misma.
El resultado de hacer esta operación quedaría como $\sigma := succ(Z * Z) = Y \implies \exists Z'.(Z * Z) + Z' = Y$
Importante: Recordar que solamente reemplazamos las ocurrencias libres de una variable, no la que está ligada a un cuantificador.

Deducción Natural extendida a LPO

Todas las reglas de deducción natural proposicional siguen vigentes pero acá se agregan las reglas de introducción y eliminación para el $\forall$ y $\exists$.
Igual que antes:

- Un contexto Γ es un conjunto finito de fórmulas.
- Un secuente es de la forma $\Gamma \vdash \sigma$

Axioma	AX		
Conjunción	$\wedge$I	$\wedge$E$_1$	$\wedge$E$_2$
Disyunción	$\vee$I$_1$	$\vee$I$_2$	$\vee$E
Implicación	$\Rightarrow$I	$\Rightarrow$E	
Negación	$\neg$I	$\neg$E	
Contradicción	$\bot$E		
Lógica clásica	$\neg\neg$E		
Cuantificación universal	$\forall$I	$\forall$E	
Cuantificación existencial	$\exists$I	$\exists$E	

Regla de eliminación

$$\frac{\Gamma \vdash \forall X.\, \sigma}{\Gamma \vdash \sigma\{X := t\}}\,\forall_{\mathrm{E}}$$

Regla de introducción

$$\frac{\Gamma \vdash \sigma \quad X \notin \mathrm{fv}(\Gamma)}{\Gamma \vdash \forall X.\, \sigma}\,\forall_{\mathrm{I}}$$

Regla de introducción

$$\frac{\Gamma \vdash \sigma\{X := t\}}{\Gamma \vdash \exists X.\, \sigma}\,\exists_{\mathrm{I}}$$

Regla de eliminación

$$\frac{\Gamma \vdash \exists X.\, \sigma \quad \Gamma, \sigma \vdash \tau \quad X \notin \mathrm{fv}(\Gamma, \tau)}{\Gamma \vdash \tau}\,\exists_{\mathrm{E}}$$

Estructuras de Primer Orden

Nos ayuda a decir **cómo interpretamos** los elementos del universo (M, I). Definimos como **estructura de primer orden** como el par $\mathcal{M} = (M, I)$

- $\mathcal{M}$: Conjunto **no vacío**, llamado universo.
- $\mathcal{I}$: Es una función que le da una interpretación a cada símbolo.
- Para cada símbolo de función **f** de aridad n: $I(f) : M^n \to M$
- Para cada símbolo de predicado **P** de aridad n: $I(P) \subseteq M^n$

¿Por qué es necesario esto? Hasta este momento $\forall X.X$ no nos dice nada. Es solo sintáxis.
Cuando hablamos de 0 podría significar falso en el ámbito o universo de los booleanos, podría significar 0 en los naturales.
Para poder darle todo este significado a cada símbolo tenemos que usar la función de interpretación. La función de interpretación

es re importante, porque algo que está escrito de la misma manera puede ser verdadero o falso según en donde habitemos.

$$M := \mathbb{N} \quad \text{(los elementos son números naturales)}$$

$$\begin{aligned} I(0) &= 0 \\ I(\mathtt{succ})(n) &= n+1 \\ I(+)(n,m) &= n+m \\ I(*)(n,m) &= n \cdot m \end{aligned} \qquad \begin{aligned} (n,m) \in I(=) &\iff n = m \\ (n,m) \in I(<) &\iff n < m \end{aligned}$$

Bajo esta estructura, la fórmula $\forall X.\, X = X + X$ es falsa.

Asignación

Una asignación es una función que a cada variable le asigna un elemento del universo: $a : \mathcal{X} \to M$

Interpretación de Términos

Cada término $t \in \mathcal{T}$ se interpreta como un elemento, extendiendo la definición de **a** a términos: $a(t) \in M$: $a(f(t_1, \ldots, t_n)) = I(f)(a(t_1), \ldots, a(t_n))$

Interpretación de Fórmulas

Suponemos fijada una estructura de primer orden $\mathcal{M} = (M, I)$.
Definimos una relación de **satisfacción** $\mathbf{a} \vDash_{\mathcal{M}} \sigma$ "La asignación **a** (bajo la estructura $\mathcal{M}$ satisface la fórmula σ)"

$$\begin{array}{lll}
\mathbf{a} \vDash_{\mathcal{M}} \mathbf{P}(t_1, \ldots, t_n) & \text{sii} & (\mathbf{a}(t_1), \ldots, \mathbf{a}(t_n)) \in I(\mathbf{P}) \\
\mathbf{a} \vDash_{\mathcal{M}} \sigma \wedge \tau & \text{sii} & \mathbf{a} \vDash_{\mathcal{M}} \sigma \ \text{y} \ \mathbf{a} \vDash_{\mathcal{M}} \tau \\
\mathbf{a} \vDash_{\mathcal{M}} \sigma \vee \tau & \text{sii} & \mathbf{a} \vDash_{\mathcal{M}} \sigma \ \text{o} \ \mathbf{a} \vDash_{\mathcal{M}} \tau \\
\mathbf{a} \vDash_{\mathcal{M}} \sigma \Rightarrow \tau & \text{sii} & \mathbf{a} \nvDash_{\mathcal{M}} \sigma \ \text{o} \ \mathbf{a} \vDash_{\mathcal{M}} \tau \\
\mathbf{a} \vDash_{\mathcal{M}} \neg\sigma & \text{sii} & \mathbf{a} \nvDash_{\mathcal{M}} \sigma \\
\mathbf{a} \nvDash_{\mathcal{M}} \bot & & \\
\mathbf{a} \vDash_{\mathcal{M}} \forall X.\, \sigma & \text{sii} & \mathbf{a}[X \mapsto m] \vDash_{\mathcal{M}} \sigma \text{ para todo } m \in M \\
\mathbf{a} \vDash_{\mathcal{M}} \exists X.\, \sigma & \text{sii} & \mathbf{a}[X \mapsto m] \vDash_{\mathcal{M}} \sigma \text{ para algún } m \in M \\
 & & \\
\mathbf{a} \vDash_{\mathcal{M}} \sigma \clubsuit \tau & \text{sii} & \mathbf{a} \vDash_{\mathcal{M}} \sigma \text{ brócoli } \mathbf{a} \vDash_{\mathcal{M}} \tau
\end{array}$$

Importante: $a[x \to m]$ representa el mapsTo y está definido de la siguiente forma:
$a : V \to M$
$a[x \to m] : V \to M$
$a[x \to m](y) = \begin{cases} m \ si \ x = y \\ a \ si \ x \neq y \end{cases}$
Véase **anexo** para ver ejemplos de fórmulas válidas, aplicaciones de predicados y símbolos de función

Decimos que una fórmula σ es:

VÁLIDA si $\mathbf{a} \vDash_{\mathcal{M}} \sigma$ para toda $\mathcal{M}, \mathbf{a}$	SATISFACTIBLE si $\mathbf{a} \vDash_{\mathcal{M}} \sigma$ para alguna $\mathcal{M}, \mathbf{a}$
INVÁLIDA si $\mathbf{a} \nvDash_{\mathcal{M}} \sigma$ para alguna $\mathcal{M}, \mathbf{a}$	INSATISFACTIBLE si $\mathbf{a} \nvDash_{\mathcal{M}} \sigma$ para toda $\mathcal{M}, \mathbf{a}$

Observaciones

σ es VÁLIDA	sii	σ no es INVÁLIDA
σ es SATISFACTIBLE	sii	σ no es INSATISFACTIBLE
σ es VÁLIDA	sii	$\neg\sigma$ es INSATISFACTIBLE
σ es SATISFACTIBLE	sii	$\neg\sigma$ es INVÁLIDA

Pre-Prolog

Términos de Primer Orden

Prolog opera con ellos. Son de la pinta X Y $succ(succ(zero))$ $bin(I, R, D)\ldots$

Fórmulas Atómicas

En Prolog son de la forma $pred(t_1, ..., t_n)$
Ej.: $padre(zeus, atenea)$

Programa en Prolog

Es un conjunto de reglas: $\sigma :- \tau_1, \ldots, \tau_n$.
Cada σ es una regla.
Ej.: $abuelo(X, Y) :- padre(X, Z),\ padre(Z, Y)$. En este caso $\sigma := abuelo(X, Y)$, $\tau_1 := padre(X, Z)$ y $\tau_2 := padre(Z, Y)$.

Hechos

Son aquellas reglas en las cuales $n = 0$. Es lo que se considera verdadero o tautológico.
¿A qué nos referimos con $n = 0$? Bueno, que no dependen de nada.
Ej.: $\sigma :- padre(zeus, ares)$

Interpretación Lógica de las Reglas

Cada uno de los τ en conjunción deben implicar σ
Es decir, tienen la siguiente interpretación lógica $\forall X_1, \ldots \forall X_k((\tau \wedge \cdots \wedge \tau_n) \implies \sigma)$ donde $X_1, \ldots, X_k$ son todas las variables libres de las fórmulas.
Ej.: $\forall X.\forall Y.\forall Z.((padre(X, Z) \wedge padre(Z, Y)) \implies abuelo(X, Y))$

Consultas

Hablamos de Consultas en Prolog cuando aparece una incógnita X.
Una consulta es de la forma: $? - \sigma_1, \ldots, \sigma_n$
Ej.: $? - abuelo(X, ares)$

Interpretación Lógica de las Consultas

El X hay que predicarlo en base a existenciales.
Ej.: $\exists X_1, \ldots, \exists X_k.(\sigma_1 \wedge \cdots \wedge \sigma_n)$ donde $X_1, \ldots, X_k$ son todas las variables libres de las fórmulas.

Claúsula y Literales

Claúsula: $(P \wedge Q)$.
Literal: P

Lógica Proposicional

Pasaje de Logica Proposicional a Forma Clausal

Es un algoritmo.
La entrada es una **fórmula σ de la lógica proposicional** y la salida es un booleano que indica si σ es válida.

- Reescribir los $\implies$: $a \implies b \equiv \neg a \vee b$
- Pasar a Forma Normal (f.n) Negada: Empujar los $\neq$ hacia adentro (si hay).
- Pasar a Forma Normal (f.n) Conjuntiva: Distrubuir $\vee$ sobre $\wedge$ (si hay).

Luego, la forma Clausal está formada por conjunciones de disyunciones.
Ej.: $(p \vee q) \wedge (q \vee r)$ en Forma Clausal es $\mathcal{C} = \{\{p, q\}, \{q, r\}\}$. En este caso, esta Forma Clausal está dado por 2 Cláusulas, donde cada Claúsula tiene 2 literales.

Prioridad de Formas Normales en Logica Proposicional

Hay tantas que puede ser un quilombo pero es así: $negada \subseteq conjuntiva$

Refutación (Método de Resolución) en Lógica Proposicional

Llamemos σ a una fórmula de la lógica proposicional cualquiera.
Refutación de $\mathcal{C}$: Derivación de $\mathcal{C} \vdash \bot$
Si encontramos una refutación de $\mathcal{C}$:

- Vale $\neg\sigma \vdash \bot$: Es decir, $\neg\sigma$ es insatisfactible o contradicción.
- Luego, vale $\vdash \sigma$. Es decir, σ es tautología.

Si NO encontramos una refutación de $\mathcal{C}$:

- No vale $\neg\sigma \vdash \bot$. Es decir, σ es satisfactible.
- Luego, no vale $\vdash \sigma$. Es decir, σ no es válida.

Es un algoritmo y tiene varios pasos pero veamos un ejemplo.
Ej.: $(P \wedge Q) \implies P$ ¿Cuáles son tautologías? Si nos está pidiendo cuales son tautologías tenemos que ver que $\neg\sigma$ sea insatisfactible o contradicción, si esto sucede entonces σ es tautología.
Lo primero que hacemos es pasar la fórmula $\neg\sigma$ a Forma Clausal, por el algoritmo descripto anteriormente nos queda

- $\neg((P \wedge Q) \implies P)$
- Eliminación del $\implies$: $\neg(\neg(P \wedge Q) \vee P)$
- Empujar $\neg$ interno: $\neg(\neg P \vee \neg Q \vee P)$
- Empujar $\neg$ externo: $P \wedge Q \wedge \neg P$

Luego, la forma Clausal es $\mathcal{C} = \{\{P\}, \{Q\}, \{\neg P\}\}$
Ahora básicamente tenemos que ir eligiendo cláusulas **k** y **k'** hasta que lleguemos al vacío.
Eligiendo $k = \{P\}$ y $k' = \{\neg P\}$ nos da $\emptyset$. Por lo tanto, se agrega $\emptyset$ a $\mathcal{C}$ pero en el paso recursivo, como $\emptyset$ pertenece a $\mathcal{C}$ llegamos a que $\neg\sigma$ es insatisfactible. Luego, σ es tautología.
Importante: Es útil ver que para llegar a vacío no siempre es necesario usar todas las cláusulas. Inclusive, va a haber veces que jamás se llegue a insatisfactible. Entonces eso indicaría que si vale $\neg\sigma$ entonces σ es inválida.
Importante: Este proceso siempre termina.

Lógica de Primer Orden (LPO)

Pasaje de LPO a Forma Clausal

Es prácticamente un algoritmo. Es parecido al anterior en los primeros 2 pasos. La diferencia es que acá si σ no es válida, el método puede no terminar.
En la teórica les ponen los pasos de una, pero a mí no me gustó como lo ordenaron. Primero lean lo de abajo, y va a haber una sección que se llama igual que esta pero con un (2).

Resolución para Lógica de Primer Orden

Hay que pasar por varias formas hasta llegar a la forma Clausal. **Importante**: Lo anoto en todos lados porque siempre me olvido. No por algo nos hacen detectar siempre las variables ligadas y libres. Si tenés más de un cuantificador con la misma variable, entonces renombrá algun cuantificador y reemplazá su ocurrencia por la nueva variable, porque después es un quilombo.

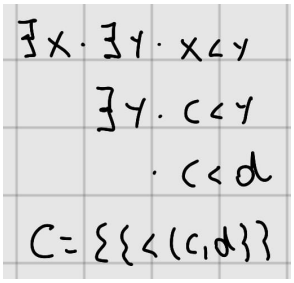
Pasaje a forma clausal en Lógica de Primer Orden

Skolemización

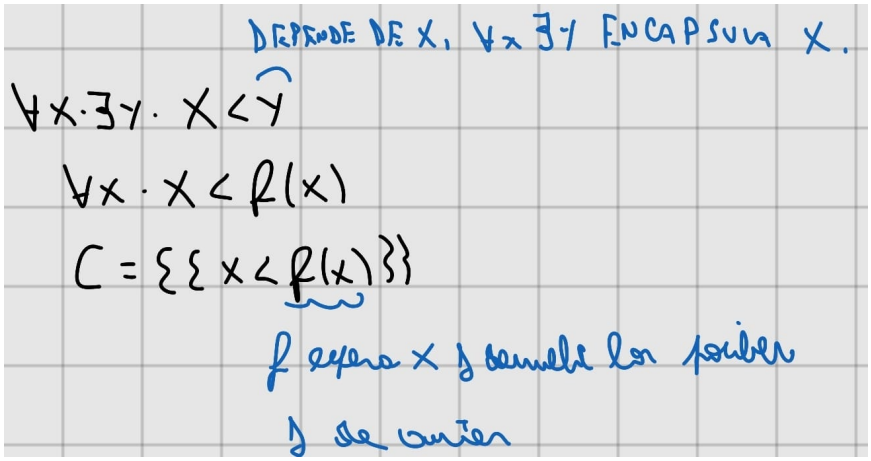
La Skolemización es deshacerse de los cuantificadores existenciales.
La idea es introducir testigos para los $\exists$ sin cambiar la satisfactibilidad.

- Todo cuantificador existencial se instancia en una constante o función de skolem.
  - Utilizamos una función cuando el cuantificador existencial está dentro al alcance de un cuantificador para todo (el más cercano).
  - Utilizamos simplemente c cuando el cuantificador existencial no está al alcance de otro cuantificador.

Importante: La Skolemización **no es determinística**.
Importante 2: Skolemizar de afuera hacia adentro.
Importante 3: La Skolemización **preserva la satisfactibilidad** pero **no produce fórmulas equivalentes**, esto quiere decir que no preserva la validez. Ej.: $\exists X.(P(0) \implies P(X))$ es válida mientras que $P(0) \implies P(c)$ es inválida.
Ejemplo de Skolemización con existenciales sin alcance de universales



Ejemplo de Skolemización con existenciales al alcance de universales



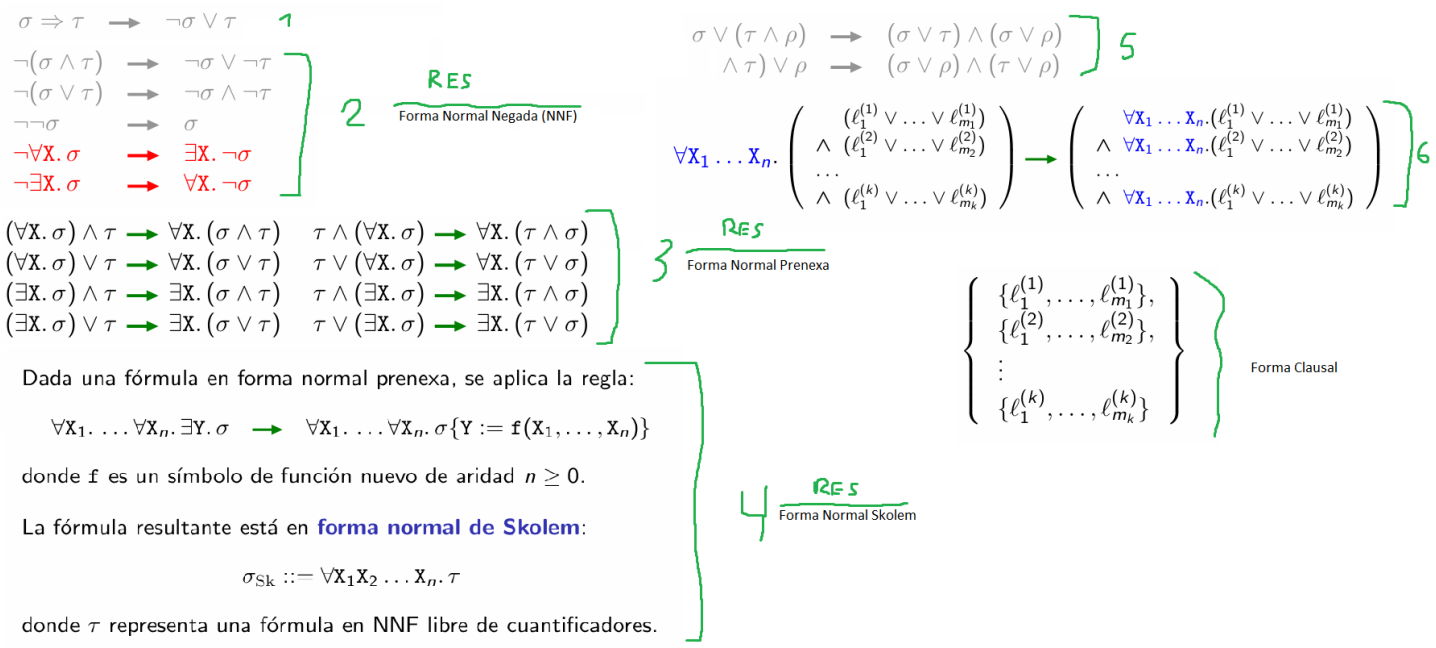
Forma Normal de Skolem (NNF)

Sea A una sentencia rectificada en FNN. Una fórmula está rectificada si todos sus cuantificadores ligan variables distintas entre sí, y a la vez distintas de todas las variables libres.
En criollo: Los cuantificadores tienen variables con nombres únicos y no se pisan con variables libres.
Una fórmula en forma normal de Skolem tiene la siguiente pinta: $\sigma_{Sk} ::= \forall X_1 X_2 \dots X_n * \tau$

Pasaje de LPO a Forma Clausal (2)

- Reescribimos $\Longrightarrow$ usando $\neg$ y $\vee$.
- Pasar a f.n negada, empujando $\neg$ hacia adentro.
- Pasar a f.n extrayendo $\forall$ y $\exists$ hacia afuera.
- Pasar a f.n de Skolem, Skolemizando los existenciales.
- Pasar a f.n. conjuntiva, distribuyendo $\vee$ sobre $\wedge$.
- Empujar los cuantificadores hacia adentro de las conjunciones.

Acá dejo una imagen que hice por si es de utilidad, con las propiedades y todo (res: es lo que te devuelve al aplicar ese paso)



Véase **anexo** para dos ejemplos

Prioridad de Formas Normales en LPO

Hay tantas que puede ser un quilombo pero es así: $negada \subseteq prenexa \subseteq skolem \subseteq conjuntiva$

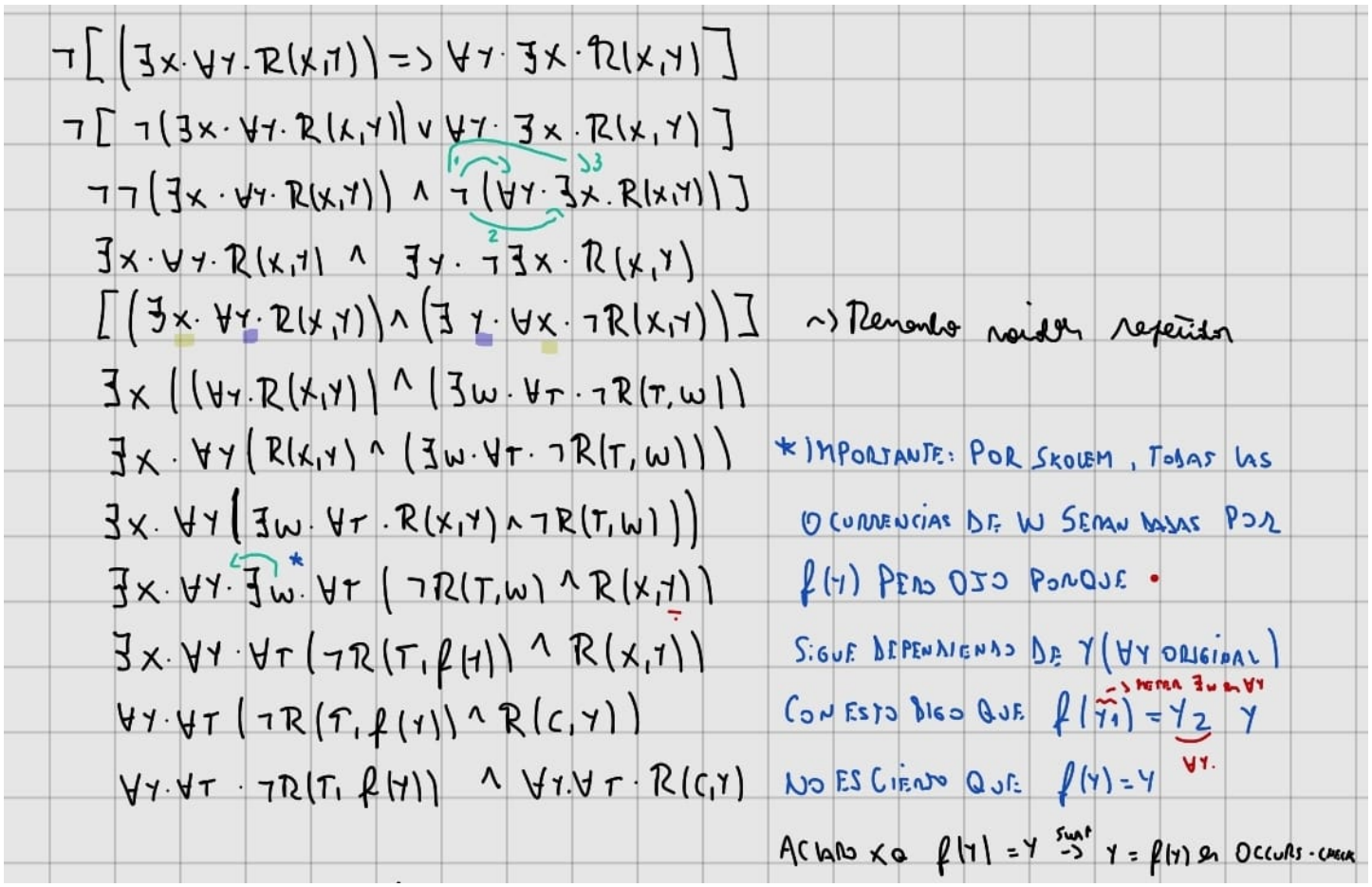
Refutación (Método de Resolución) en LPO

Es exactamente igual que en Lógica Proposicional **excepto** que ahora como existen funciones P que tienen argumentos, hay que ver si los argumentos son unificables.
Hasta ahora, esto vale $\forall X.P(X) \wedge \neg P(0)$, esto es porque las claúsulas que tenemos son $\mathcal{C} = \{\{P(X)\}, \{\neg P(0)\}\}$ y la regla propuesta no aplica porque las P son iguales pero sus argumentos no.
Ej.: Sea $\neg\sigma$ con sus cláusulas, $\mathcal{C} = \{\{P(X,Y)\}, \{\neg P(f(z),Z)\}\}$

- Para cancelar los términos opuestos P y $\neg P$, unificamos.
- $mgu(P(X,Y) \stackrel{?}{=} P(f(Z),Z))$. Para que estos sean iguales en argumento vemos que unificamos como $\{X := f(z), Y := Z\}$.
- Luego, k = $P(f(z),Z)$ y k'= $\neg P(f(z),Z)$ nos da como resultado $\{\}$ o $\emptyset$.
- Por lo tanto como $\neg\sigma$ es insatisfactible, σ es tautología.

Importante: Primer paso de todo ejercicio: si comenzamos teniendo implicaciones, o cuantificadores universales existenciales con las mismas variables, recordar renombrarlas.
Importante2: (A nivel MGU) Si hay un cuantificador existencial que ahora depende un universal, primero hay que chequear **en qué términos** se usa la variable **ligada al cuantificador universal**. Esto es, porque cuando eliminemos el **cuantificador existencial**, la existencial se elimina y será del tipo **f(variableCuantUniversal)** pero como variableCuantUniversal estaba en otro término (no ligado al existencial) al intentar unificar nos dará occurs-check pues variableCuantUniversal NO unifica con

f(variableCuantUniversal). La idea justamente acá, es que como el existencial asegura que existe (puede no existir), entonces no podemos confundir con la ligada al para todo porque esa sí vale siempre. Igual, dejo un ejemplo acá abajo.



OJO. Esto es SOLO a nivel MGU. Con esto quiero decir que si vas a unificar, le cambies el nombre a las variables.

Prolog

El algoritmo que vimos antes para ver si una fórmula de primer orden σ es válida funciona, pero es costosa. Esto es porque a nivel computacional implica hacer

- Búsqueda: Elegir dos claúsulas
- Selección: Elegir un subconjunto de literales de cada claúsula

Además en cada paso se agrega una claúsula, resolverse ecauciones de unificación y se usa BFS.
¿Cómo soluciona este problema Prolog? Usando **resolución SLD**.

Resolución SLD

La resolución SLD se aplica solamente sobre cláusulas de Horn.

Cláusulas de Horn

Recordemos que una claúsula es un conjunto de literales

$$\{l_1, \ldots, l_n\}$$

Los literales son de la forma

$$l ::= P(t_1, \ldots, t_n) \mid \neg P(t_1, \ldots, t_n)$$

En SLD cada cláusula tiene como máximo un literal positivo.
Por ejemplo, en SLD no es posible resolver esto: $\mathcal{C} = \{\{P(X), Q(Y)\}, \{\neg Q(Y)\}\}$ porque la primera claúsula tiene dos positivos.

Grupos en Claúsulas de Horn

- Las que tienen un literal positivo y ningún negativo se llaman hechos o axiomas.
  - En LPO: $\{P(X)\}$
  - padre(m, t): Es un axioma que dice que m es padre de t.
  - madre(k, t): Es un axioma que dice que k es madre de t.
- Las que tienen un literal positivo y una/varias negativas se llaman reglas.
  - En LPO: $\{P(X), \neg Q(X, Y)\}$
  - En Prolog: padres(x) :- madre(m, x), padre(p, x).
  - Esta regla define que x tiene 'padres'cuando existe una madre m de x y un padre p de x.
- Las que NO tienen literal positivo y una/varias negativas se llaman cláusulas objetivo.
  - En LPO: $\{\neg P(X)\}$
  - padres(t).
  - Cuando esto le llega a Prolog, se conforma una claúsula objetivo que se compone de literales negativos.

Nota: Los axiomas + reglas en Prolog se llaman **cláusulas de definición (PGM)** y estas nunca llegan a ser insatisfactibles.

Derivación SLD

En cada paso, Prolog siempre toma una **claúsula de definición (PGM)** y una **claúsula objetivo (Goal)**

Programación Orientada a Objetos (POO)

No subieron la teórica así que voy a tener que escribir en base a lo que anoté yo y me pareció interesante.
Vamos a usar SmallTalk donde todo es prácticamente un objeto.
En SmallTalk los objetos se comunican entre sí a través de mensajes, donde cada mensaje es una invocación a un método de un objeto.
Como en el Paradigma Orientado a Objetos podemos mutar instancias de objetos determinados a través de métodos (getters/setters), necesitamos tener un estado. Al principio de la materia vimos que el estado no es más que el conjunto o estado de variables en un momento dado.

Mensajes

Siempre están asociados a un objeto.

Métodos de Instancia y Métodos de Clase

Los métodos no son más que la implementación de la respuesta a un mensaje dado.

```
Object subclass: Persona [
    Persona class >> hola [
        ^'hola, como estas'
    ]

    Persona class >> crearNuevaPersona [
        ^Persona new.
    ]
]

obj := Persona new.
obj hola.
```

En este caso el mensaje es hola (invocar al método de instancia hola), y la respuesta 'hola, como estas'la devuelve el método hola de la clase persona.
La diferencia entre un método de instancia y método de clase es la siguiente

- Método de Instancia: Son métodos que se pueden ejecutar **a través de la instancia de una clase**.
  - Ej.: obj hola.
- Método de Clase: Se ejecutan en la clase misma, y nos pueden ayudar a la creación de nuevas instancias.
  - Ej.: obj := Persona crearNuevaPersona.

Objeto Receptor

Es el objeto que recibe el mensaje. Puede recibirlo tanto una clase a través de un método de clase, o una instancia de la clase a través de un método de instancia.

Objeto Colaborador

Es un objeto con el que el receptor interactúa o colabora para lograr una tarea o propósito.
Es un parámetro. Ej.: 10 mcm: 4, 4 actúa como colaborador.

Return

En el código, el return se puede notar como $\wedge\ variable$

Uso de Variables en Clases

No nos referimos a ella como this, como si hacemos en muchos lenguajes, sino que acá, en SmallTalk simplemente mencionamos la variable.

POO y Clean Code

Jamás, bajo ningún termino agregamos campos a una clase por fácilidad de cómputo. Las variables que representan a una clase deben ser la menor cantidad, y deben estar pensadas para el objetivo que tenga ese objeto de resolver.
Esto es más que nada porque, un objeto debería ser lo más chico posible además de que si agregamos campos extra, hay que mantenerlos y eso da un costo mayor.

Tipos de Expresiones en SmallTalk

- Variables Locales: Se escriben en minúscula
  - x:=3. y:=2.
- Variables Globales: Se escriben en mayúscula (no obligatorio). Se definen a nivel de clase, y son accesibles desde cualquier parte del programa a través de la clase a la que pertenecen.
  - classVariableNames: 'contador'
- Mensajes Unarios: Toman un único parámetro. Se usan de manera prefija.
  - 10 print. Imprime el valor 10.
  - 5 isOdd. Devuelve true.
  - -1. Devuelve el número -1, el (-) es un mensaje unario.
- Mensajes Binarios: Toman dos parámetros. Se usan de manera infija.
  - 3 + 5.
  - 10 * 2.
  - true and: false.
- Mensajes Keyword: Pueden tener múltiples argumentos, con cada argumento despues de su palabra clave correspondiente.
  - Rectangulo new ancho: 5 alto:10. Los keywords son ancho y alto.
- Asignación
  - persona := Persona new

La prioridad, dentro de una expresión es la siguiente: $unarios > binarios > keyword$
Veamos un ejemplo donde tiene importancia esto: Sea una clase de un entero, implemente un método de instancia que permita saber cual es el mcm del número instanciado y otro que viene por parámetro.

```
1    mcm: numberTwo
2        | res |
3        res := self * b // self gcd: b
4        ^res
```

Como está implementado este código va a fallar, porque primero va a hacer self * b, e inmediatamente va a querer dividir por algo que todavía no se resolvió.

```
mcm: numberTwo
    | res |
    res := self * b // (self gcd: b)
    ^res
```

Importante: Nótese que todos los métodos los estamos haciendo dentro de una clase. Esto quiere decir que siempre tenemos que asumir que estamos dentro de un objeto.
Si tuviesemos que notar qué mensajes, receptores, colaboradores, resultados hay para la siguiente entrada: $6\ mcm\ 10$

- Mensaje: mcm — Receptor: 6 — Colaboradores: 10 — Resultado: 30
- Mensaje: * — Receptor: Self (6) — Colaboradores: 10 — Resultado: 60
- Mensaje: gcd — Receptor: Self (6) — Colaboradores: 10 — Resultado: 2
- Mensaje: // — Receptor: 60 — Colaboradores: 2 — Resultado: 30

Nótese que básicamente fuimos paso a paso viendo qué instrucciones estaban y quienes estaban involucrados.

Palabras Reservadas en SmallTalk

- true
- false
- self
- super
- nil
- thisContext

Algunos ejemplos de instancias de objetos

- nil: Es la instancia de UndefinedObject
- true: Es la instancia de la clase True
- false: Es la instancia de la clase False
- Constantes Númericas: 1, 12, 125... son números, instancias de clases como SmallInteger o LargeInteger.
- Strings: $'hola'$ es la instancia de la clase String
- Símbolos: #Rectángulo. Instancia de la clase Symbol. Un Símbolo es un objeto que representa un identificador único e inmutable.
- Caracteres: $a es la instancia de la clase Character. Representa un único caracter.

Herencia

Importante: En la materia no se lo menciona, pero siempre es mejor priorizar la composición antes que la herencia.
La Herencia trae muchos problemas a nivel de objetos porque se vuelve inmantenible si la clase padre tiene muchos atributos, y esto es importante, porque una clase no debería estar abierta a eliminar cosas núcleo pero sí abierta a extenderla.
Esto es porque si refactorizamos la clase padre, si hay mas de una clase que heredaban esta, vamos a tener que considerar modificar andá a saber cuantas más clases.
El concepto de Herencia es importante, porque cuando hablamos de una relación padre-hijo, si desde hijo queremos llenar campos que están en el padre, usamos la palabra super, mientras que si queremos hablar de nuestro ámbito local de objeto usamos self.
Voy a mostrar un ejemplo en un lenguaje que conozca bien (TypeScript) para que después puedan relacionarlo en cualquier lenguaje.

```
class ContactoBasico {
    msg: string;

    constructor(data){
        this.msg = data.msg;
    }
}
```

```
8
9     class ContactoEmail extends ContactoBasico {
10        email: string;
11
12        constructor(data){
13            super(data);
14            this.email = data.email;
15        }
16
17        hacerAlgo(){
18            console.log(‘Email: ${this.email}, Mensaje: ${this.msg}‘);
19        }
20    }
```

En el ejemplo anterior, podemos ver que ContactoEmail puede almacenar un valor en msg como si fuese una variable de clase de sí misma, pero en realidad, esta puede ser utilizada a nivel ContactoEmail porque la estamos llenando vía super.

Self

Cuando hablamos de Self, estamos hablando de utilizar las variables o métodos de la instancia del objeto que estamos manipulando.
El Self siempre busca hacia abajo.

Super

Cuando hablamos de Super, estamos hablando de herencia. Esto quiere decir que si una clase habla de super, está extendiendo de otra. Ojo, super habla de la clase inmediatamente que está por encima.
Esto es súper importante, porque hay muchos lenguajes (y gracias a dios) que no aceptan herencia múltiple.

```
1     class ContactoBasico {
2         msg: string;
3
4         constructor(data){
5             this.msg = data.msg;
6         }
7
8         hacerAlgo(){
9             console.log("Hola");
10        }
11    }
12
13    class ContactoEmail extends ContactoBasico {
14        email: string;
15
16        constructor(data){
17            super(data);
18            this.email = data.email;
19        }
20
21        hacerAlgo(){
22            console.log(‘Email: ${this.email}, Mensaje: ${this.msg}‘);
23        }
24    }
25
26    class ContactoEmailMasComplicado extends ContactoEmail {
27        emailSecundario: string;
28
29        constructor(data){
30            super(data);
31            this.emailSecundario = data.emailSecundario;
32        }
33
34        hacerAlgo(){
35            return super.hacerAlgo();
```

```
        }

    }
```

Ojo. En este ejemplo, si queremos instanciar ContactoEmailMasComplicado tendremos que cumplir la precondición de super, es decir, el constructor de ContactoEmail, y a su vez el de ContactoBásico. Esto es porque es una cadena. Ya se dan cuenta del por qué es malo laburar con herencia.
¿Qué es lo que sucede, si una vez instanciada ContactoEmailMasComplicado en obj, llamamos obj.hacerAlgo? ¿Qué hacerAlgo ejecuta?, bueno, el padre inmediato. Ejecutaría console.log('Email: $this.email, Mensaje$:this.msg'), es decir, el hacerAlgo() que está en ContactoEmail, que es padre de ContactoEmailMasComplicado.

Importancia del Return

En los métodos de una clase, si no tenemos un return, entonces el método retornará **self**. Es decir, la instancia.

Métodos de Clase

Los métodos de clase nos permiten crear instancias de una clase sin utilizar desde fuera de la clase la palabra reservada **new**. Esto nos permite que la clase por sí misma, diga qué es lo que podemos usar para crear instancias de ella.

```
Object subclass: #Rectangulo
        instanceVariableNames: 'ancho␣alto'
        classVariableNames: ''

Rectangulo class >> ancho: ancho alto: alto [
    ^self new
        ancho: ancho;
        alto: alto;
        yourself.
]

Rectangulo >> ancho: valor [
    ancho := valor
]

Rectangulo >> alto: valor [
    alto := valor
]

Rectangulo >> descripcion [
    ^'Rectángulo␣de␣ancho␣', ancho printString, '␣y␣alto␣', alto printString
]

Uso:
rect := Rectangulo ancho: 30 alto: 20.
rect descripcion.
```

Importante notar, que cuando hacemos $\wedge self\ new\ ancho : ancho$; estamos creando una nueva instancia, mandando un mensaje al método ancho con el parámetro ancho.
Importante notar también, que cuando hacemos self new, estamos enviando un mensaje de new con varios argumentos. Estos argumentos los separamos por ; porque son parte del mismo mensaje.

Sobrecarga

Decimos que un método de clase está sobrecargado cuando existe más de un método con el mismo nombre pero que toman diferentes parámetros (ya sea en cantidad o en tipos, lo importante es que no sean la misma cantidad de parámetros en el mismo orden y mismo tipo). Es importante notar que los métodos de clase más restrictivos deberían poder estar encapsulados en el más general.

```
object subclass #robot
       instance variablenames: 'posicion'

       inicializar: unaposicion
          posicion := unaposicion

```

```
        posicion:
           ^posicion

         inicializar:
             self inicializar: (0 @ 0)
```

Sintaxis de SmallTalk

- Usá **.** para indicar donde termina una instrucción. Es similar al **;** de C++ pero acá, si hay solamente una línea el **.** no es necesario.
  - Ej.: x := 10. y := 20.
- Usá **;** cuando tengas que encadenar mensajes a un mismo objeto, sin finalizar la ejecución de la expresión anterior.
  - Ej.: Transcript show: 'Hola'; cr; show: 'Mundo'

Bloques (Block-Closures)

Permite encapsular lógica dentro de un código independiente y reutilizable. Está compuesto por una secuencia de instrucciones que se agrupan y pueden ser pasadas como argumentos a métodos o invocadas como acción dentro de un método.
La sintáxis está compuesta por :variable + expresión
Si solo hay que definirlo, lo terminamos con **.**

```
    [:x | x * 2]
```

Si hay que pasarle argumentos, lo invocamos con **:**

```
    resultado := (1 to: 3) collect: [:x | x * 2].
    resultado tiene [2, 4, 6]
```

Si hay que obtener el resultado de un bloque, colocamos value antes del **.**.
El value invoca al bloque. Un value vacío es llamar al método sin argumentos.

```
    resultado := [1 + 2 + 3] value.
    resultado tiene el valor de 6
```

Si hay que usarlo en condicionales, cada rama podría ser un bloque

```
    resultado := 10 > 5
        ifTrue: ['es␣mayor']
        ifFalse: ['es␣menor'].
```

Si hay que enviarle argumentos a los bloques, los enviamos a través de value, es decir, invocamos al bloque con tantos argumentos instanciados.

```
    resultado := [:x :y | x+y ] value: 3 value: 4.
```

Si querés usar variables locales en los bloques, la sintaxis es diferente

```
    resultado := [ | x y | x:= 3. y:=2. x+y. ]
```

Importante: Si un bloque está definido para n argumentos, solo se podrá invocar ese bloque enviando los n argumentos.

```
    resultado := [:x :y | x+1] value: 1. falla
    resultado := [:x :y | x+1] value: 1 value: 3. funciona
```

Para invocar bloques anidados, simplemente usamos value reiteradas veces pero indicando la cantidad de argumentos.

```
    [ | z | z:=10 . [:x | x+z]] value value: 10.
    Invoca al bloque padre, y como el segundo bloque espera un argumento obligatorio le enviamos el valor
         de 10.
```

Importante: **NUNCA** usar un return dentro de un bloque. Es algo bastante oscuro.
Importante: Los bloques devuelven como resultado la última expresión del bloque.

Bloques = Block Closures

¿Qué relación tienen los bloques con los closures? Cuando guardamos un método que contiene un bloque, y a su vez, el bloque utiliza una variable de ese método, estamos encapsulando a este bloque en un universo donde las variables que se usen van a ser inmmutables, a menos que ejecutemos este bloque reiteradas veces.
Veamos un ejemplo

```
A m1
    | x y |
    y := 0
    x := [ y := y+1 ].
    ^x

B m2
    | a aBlock anotherBlock |
    a := A new.
    aBlock := a m1.
    aBlock value.
    aBlock value.
    anotherBlock := a m1.
    anotherBlock value.
    ^aBlock value.
```

Prestemos atención a A por un momento, A es un método que básicamente define dos variables locales **x, y** donde **y** es inicializada con 0, y **x** es inicializada con un bloque que depende de **y**. Al finalizar el método, devuelve el bloque.
Ahora vayamos a B m2. La variable **a** almacenará la instancia de A creada (para poder usar el método de instancia m1). Luego que hace esto hay dos partes, una variable llamada aBlock, donde aBlock almacenará el block-closure [y := y+1].
Luego, la instrucción aBlock value ejecuta el closure esperando un valor, y hasta ahora, el valor que arroja es 1 (Nótese que arroja 1 porque es 0 := 0+1)
Luego, la instrucción aBlock value, ejecuta el closure esperando un valor, y hasta ahora, el valor que arroja es 2 (Nótese que arroja 2 porque en la llamada anterior, y valía 1, entonces 1 := 1+1 = 2)
Lo interesante viene ahora, anotherBlock := a m1. almacena el bloque de m1 pero como una instancia NUEVA. Es decir, es otro bloque INDEPENDIENTE.
Por lo tanto cuando hagamos anotherBlock value. arrojará y = 1.
Finalmente, aBlock value sumará 1 al primer bloque, y retornará su valor, es decir, y = 3. **Conclusión**: Ninguno de los bloques sabe que están instanciados para variables diferentes, y cada bloque recuerda qué valor tomaba la variable y en cada instrucción.
¿Cuál es el mensaje, cual el receptor y cual sus colaboradores? bueno, el mensaje sería m1, el receptor sería el propio objeto a y no hay ningún colaborador.

Excepciones

Si mandamos un mensaje a un método que no existe en la clase, se tratará de buscar en cada uno de los super recursivamente. El último intento lo hará en el padre de todos los objetos, es decir, la clase Object. Si no existe ahí, se ejecutará una excepción (does not understand).

Colecciones en SmallTalk

Se indexan desde el 1 en adelante. Les querían caer mal a los que indexamos desde 0.
Algunas de las colecciones mas conocidas son

- Bag (Multiconjunto)
- Set (Conjunto)
- Array (Arreglo): La cantidad de elementos es fija.
- OrderedCollection (Lista)
- SortedCollection (Lista ordenada)
- Dictionary (Hash)

El mensaje que hay que utilizar para crear estas colecciones es **with**.
Hay diferentes maneras (equivalentes) de crear una colección y agregar sus elementos

- Bag with: 1 with: 2 with: 4
- # (1 2 4) = (Array with: 1 with: 2 with: 4) →. Ojo, el # es un símbolo, por lo tanto sería una lista inmutable.

- Bag withAll: #(1 2 4)

Algunos mensajes que aceptan las colecciones

- add: agrega un elemento.
- at: devuelve el elemento en una posición.
- at:put: agrega un elemento a una posición.
- includes: responde si un elemento pertenece o no.
- includesKey: responde si una clave pertenece o no.
- do: evalúa un bloque con cada elemento de la colección.
- keysAndValuesDo: evalúa un bloque con cada par clave-valor.
- keysDo: evalúa un bloque con cada clave.
- select: devuelve los elementos de una colección que cumplen un predicado.
- reject: la negación del select.
- collect: devuelve una colección que es resultado de aplicarle un bloque a cada elemento de la colección original.
- detect: devuelve el primer elemento que cumple un predicado.
- reduce: toma un bloque de dos o más parametros de entrada y hace fold de los elementos de izquierda a derecha.

Anexo

Prolog

Un programa en Prolog está compuesto por

- Hechos: no poseen un cuerpo. Son verdaderos siempre.
- Reglas: tambien llamados predicados. El lado izquierdo de una regla es conocido como HEAD mientras que el lado derecho, o las condiciones que deben de cumplirse para que valga el HEAD se llama BODY.
- Consultas

Los Hechos y las Reglas conforman una database o cláusulas.
La idea de Prolog consiste en realizar consultas, es decir, preguntar cosas acerca de la información que tenemos almacenada.

Hechos

Comienzan con minúscula siempre, y están en lowercase.

```
1  woman(mia).
2  woman(judy).
3  playsAirGuitar(judy).
4  party.
```

En este caso tenemos 4 hechos, que sin importar la situación que estemos, siempre serán verdad.
Si hacemos la siguiente consulta: **?- woman(mia)** la respuesta será true.
Si hacemos la siguiente consulta **?- woman(guada)** la respuesta será false, pues la database no tiene información acerca de que guada sea una mujer. Esto es súper importante en la Programación Lógica. Que sea falso significa que estamos en un mundo cerrado, por lo tanto, lo que no son hechos ni reglas, **siempre es falso**.

Reglas

Las reglas están conformadas por hechos en su cuerpo. El lado izquierdo de una regla es llamado HEAD mientras que el lado derecho de una regla es llamado BODY.

```
1  happy(yolanda).
2  listen2Music(mia).
3  listen2Music(yolanda).
4  playsAirGuitar(mia) :- listens2Music(mia).
```

En esta database tenemos 3 hechos y 1 regla.
Las reglas se leen como: **playsAirGuitar(mia) es verdadero si listens2Music(mia) es verdadero**. Esto es importante, porque lo que quiere decir es que cada una de las condiciones del BODY implican el HEAD.
¿Cómo es que Prolog deduce el resultado de **playsAirGuitar(mia)**? Lo primero que hace es buscar esta ecuación como un hecho. Como NO es un hecho, entonces se fija qué condiciones deben cumplirse para que esto sea verdadero (por defecto, es falso). Como observa que depende de listens2Music(mia), observa si esto es un hecho. Como efectivamente es un hecho, concluye que esto es verdadero.

Reglas $\vee$

Las Reglas $\vee$ se caracterizan por ser verdaderas si las condiciones no necesariamente se deben cumplir al mismo tiempo.
Para poder definir un $\vee$ en Prolog, usamos **;**.

```
playsAirGuitar(butch) :- happy(butch); listens2Music(butch).
```

playsAirGuitar(butch) será verdadero sí y solo sí happy(butch) es verdadero **o** listens2Music(butch) es verdadero.

Variables

Las variables en Prolog no son como en un lenguaje imperativo que significan un nombre para un valor dado. Acá las variables se unifican a valores y estas comienzan con una letra mayúscula.
Esto quiere decir que, Prolog por cada vez que trata de devolver una respuesta, en realidad está devolviendo una unificación para la cual una condición es cierta (hecho)

```
woman(mia).
woman(judy).
woman(yolanda).
loves(vincent, mia).
loves(marsellus, mia).
loves(pumpkin, honey_bunny).
loves(honey_bunny, pumpkin).
```

De la siguiente database, podemos hacer consultas como: **woman(X)**, ¿qué es lo que estamos preguntando? **dame todas las posibilidades para X donde X cumpla woman**.
Esto se resuelve de la misma manera en que nosotros hacíamos el Algoritmo de Martinelli-Montanari, quiero decir que:

- $\{woman(X) \stackrel{?}{=} woman(mia)\}$
  - Dec $\rightarrow \{X \stackrel{?}{=} mia\}$
  - Del $\rightarrow \emptyset$
  - Luego, X=mia.

Esto mismo sucede para woman(judy) y woman(yolanda).
¿Qué pasa con loves?, bueno, falla por clash porque los predicados no son el mismo, ni tampoco necesitan la misma cantidad de argumentos.

- $\{woman(X) \stackrel{?}{=} loves(vincent, mia)\}$
  - $\rightarrow$ clash.

Importante: Recordar que una variable unifica con cualquier cosa.
Veamos ahora un último ejemplo, ¿qué pasa con loves(marsellus, X)? Bueno, buscará en la database los hechos o reglas que hagan match con esta consulta, viendo qué posibles valores puede tomar X para ser verdadero.
Desde ya, no va a unificar nunca con woman porque falla por clash.
Tampoco unifica con loves(vincent, mia) por el mismo motivo

- $\{loves(marsellus, X) \stackrel{?}{=} loves(vincent, mia)\}$
  - Dec $\rightarrow \{marsellus \stackrel{?}{=} vincent\}, \{X \stackrel{?}{=} mia\}$
  - $\rightarrow$ clash entre marsellus y vincent, son dos funciones de aridad 0 y son diferentes.

Términos

En Prolog, los términos están conformados por: atómos, números, variables y términos complejos.

- Los átomos son aquellos que comienzan con minúscula, están conformados por letras minúsculas, letras mayúsculas, dígitos y el símbolo de _. Ej.: listens2Music, playsAirGuitar, 'vincent'.
- Los números son como conocemos en todos los lenguajes de programación, números.
- Las variables empiezan con mayúscula, están conformadas por mayúsculas, minúsculas, números y _ y usamos _ para hablar de una variable anónima.
- Los términos complejos o también conocidos como estructuras, están conformados por una palabra y una secuencia de argumentos. Ya vimos anteriormente algunos, por ejemplo playsAirGuitar(jody) y se nota playsAirGuitar/1 pues requiere de un argumento.

Importante: Cuando decimos que playsAirGuitar requiere de un solo argumento, decimos que es de aridad 1 o función unaria.
Esto de la aridad es útil entenderlo porque Prolog nos permite tener un mismo nombre de predicado, pero con diferente cantidad de argumentos. Esto, a bajo nivel lo trata como si fuesen dos predicados totalmente diferentes.
De ahí viene que esto es válido:

```
loves(tom, guada).
loves(tom, kari).
loves(kari, tom, andy).
```

¿Cuándo usar variables anónimas, y cuando no?

Veamos el siguiente ejemplo

```
gives_footmassage(X, mia).
kills(marsellus, Y) :- gives_footmassage(Y, mia).

gives_footmassage2(_, mia).
kills2(marsellus, Y) :- gives_footmassage2(Y, mia).
```

¿Cual es la diferencia entre $gives_footmassage$ y $gives_footmassage2$? que $gives_footmassage2$ será verdadero siempre y cuando el segundo argumento sea mia, pero jamás sabremos con certeza quién le da un footmassage a mia. Entonces, marsellus no sabe. En el $gives_footmassage$, marsellus sabe que tendrá que matar al X que le haga masajes en los pies a mia.
Entonces: NO usar variables anónimas en definiciones de predicados a menos que nunca se quiera unificar o devolver una respuesta concreta de ese argumento.

Unificación

Es **fundamental** entender qué es la unificación. Decimos que dos términos unifican si son el mismo término o ellos contienen variables que pueden ser instanciadas como términos que pueden terminar siendo términos iguales.
Unifican: 42=42 unifica porque son el mismo número, mia=mia unifica porque son el mismo átomo, X=X unifican porque son la misma variable y woman(mia) = woman(mia) unifican porque son el mismo término complejo.
No unifican: 42=41, ni tampoco woman(mia) = woman(vincent), ni tampoco loves(a, b) = loves(mia, vincent)
¿Qué sucede con woman(mia) = woman(X)? esto si unifica, porque existe una unificación para X de tal manera que woman(mia) = woman(X) y esto sucede sí y solo sí X = mia, es decir, X debe instanciarse con el valor de mia para unificar.
¿Qué sucede con loves(vincent,X) = loves(X, mia)?, esta no unifica. ¿por qué?

- Dec $\rightarrow$: $\{vincent \stackrel{?}{=} X\}$, $\{X \stackrel{?}{=} mia\}$
- Swap $\rightarrow$: $\{X \stackrel{?}{=} vincent\}$, $\{X \stackrel{?}{=} mia\}$
- Delete $\{X := vincent\}$: $\{vincent \stackrel{?}{=} mia\}$ y esto es falso, pues los términos vincent y mia no unifican.

Usualmente, no estamos solamente interesados en el hecho de que dos términos unifiquen, sino que tambien queremos ver cómo las variables tienen que ser instanciadas para que sean iguales. Prolog nos da esta información pues cuando unifica dos términos realiza todas las instanciaciones necesarias.

```
2 = 2 unifica
'2' = 2 no unifica
mia = X unifica, X = mia
X = Y unifica
loves(X,X) = loves(marsellus, mia) no unifica, porque no hay forma de que X tenga dos posibles
    unificaciones a la vez.
```

Importante: Por defecto, Prolog no tiene activo el occurs-check. Esto quiere decir, que podríamos tener referencias circulares infinitas si no tenemos cuidado. Esto quiere decir que esto: $X_1 \stackrel{?}{=} X_3 \rightarrow X_1$ es posible, aunque no debería de serlo.

Occurs-Check

El Occurs-Check es la ocurrencia de una variable que estamos definiendo, que dependa de sí misma. Esto produce referencias circulares infinitas.
Ej.: $father(X) := X$ es una regla que tiene Occurs-Check, es decir, se cuelga. Esto viene a que si $X = father(X)$ entonces esto sería father(father(X)) = father(X) pero a su vez X es father(X) entonces $X = father(father(father(father(\dots))))$
Esto sucede porque Prolog es un lenguaje optimista. Este asume que vos no le vas a enviar nada peligroso.

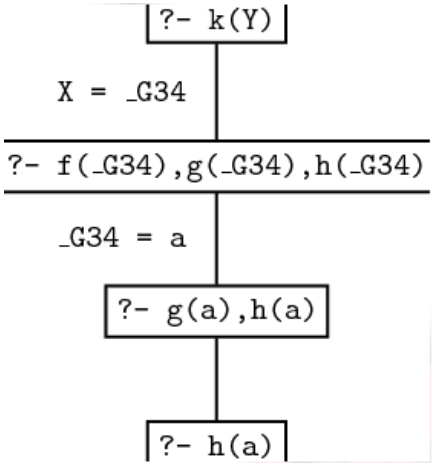
Búsqueda de Soluciones (Proof Search)

Evalúa cada una de los hechos/reglas que hagan match, luego, si la regla posee un body, va tomando cada una de las condiciones de izquierda a derecha. Busca las posibles soluciones y hace backtracking para ver cuales cumplen la segunda, y así sucesivamente. Cuando hacemos una consulta, las cosas que deben cumplirse para que sea verdadera, se llaman **goals** o claúsulas objetivo.

```
1. f(a).
2. f(b).
3. g(a).
4. g(b).
5. h(b).

6. k(X) :- f(X), g(X), h(X).
```

Desarrollemos, en forma de árbol qué es lo que hace Prolog para encontrar la solución a $? - k(Y)$

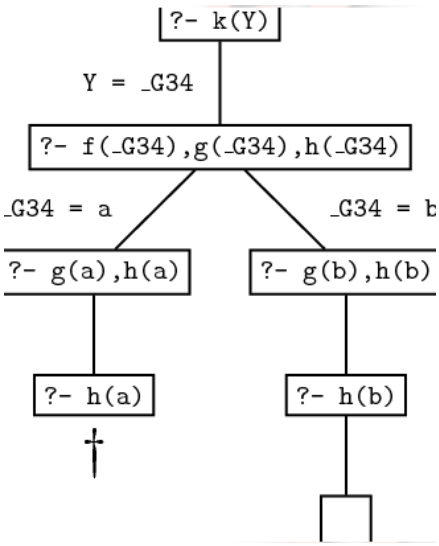
- Teniendo $k(Y)$ lo primero que hará, es buscar la primera ecuación con la que pueda unificar. como k(Y) falla tratando de unificar con los hechos 1 al 5 por Clash, entonces llegamos a que k(Y) debe unificar al HEAD de la regla k(X) :- f(X), g(X), h(X). Cuando Prolog unifica a una variable en un hecho o una regla, genera una nueva variable del estilo **k(_G34)**, entonces ahora Prolog sabe que $k(\_G34) : -f(\_G34), g(\_G34), h(\_G34)$.
- El siguiente paso es que Prolog reemplaza la consulta original $k(\_G34)$ por la lista de **goals**: $f(\_G34), g(\_G34), h(\_G34)$
- El siguiente paso es que para resolver un goal: **toma la regla que está más a la izquierda**. Para resolver el goal de $f(\_G34)$ se busca en la database, con qué hecho o regla unifica.
  - La primera unificación posible de $f(\_G34)$ es con $f(a)$, por lo tanto ahora nos quedan dos **goals** restantes. Entonces, unificamos $f(\_G34)$ a $f(a)$ y ahora, todas las ocurrencias de $\_G34$ se instancian como a.
  - Por lo tanto, los goals quedaron así: **g(a), h(a)**
  - Como g(a) es un hecho en nuestra database, nos queda solo una regla para probar, pero notamos que h(a) no existe como hecho, por lo tanto, Prolog decide que produjo un error, por lo tanto da un paso atrás y se fija qué caminos alternativos tiene para tomar antes de fallar (en este caso ninguno). Este proceso de vuelta hacia atrás es conocido como **backtracking**.



- En este momento, como Prolog se equivocó instanciando $\_G34$ con el valor de a hace un **REDO**.
- Ahora $f(\_G34)$ toma el valor de $f(b)$ (siguiente ecuación en la lista de hechos)
  - Todas las ocurrencias de $(\_G34)$ se instancian como b.
  - Por lo tanto, los goals quedaron así: **g(b), h(b)**
  - Como g(b) es un hecho, entonces el último goal por ver es h(b).
  - Luego h(b) también es un hecho.

- Por lo tanto, el conjunto de Goals quedó vacío, por lo tanto, la consulta original es satisfactible, y Prolog encontró una forma para que lo sea (instanciar Y por b). Si escribimos ;, Prolog hará backtrack y tratará nuevamente de buscar más posibles soluciones, pero en este caso, como no hay más devolverá False.

El árbol de búsqueda quedó de la siguiente manera:



Recursión en Prolog

Los predicados pueden ser definidos recursivamente. Hablando de una manera más formal, un predicado es recursivo sí y solo sí está definido de una o más reglas que refieren a sí mismo.
Los predicados recursivos están conformados por un caso base, y un caso recursivo.

```
1  is_digesting(X,Y) :- just_ate(X,Y).
2  is_digesting(X,Y) :-
3      just_ate(X,Z),
4      is_digesting(Z,Y).
5
6  just_ate(mosquito,blood(john)).
7  just_ate(frog,mosquito).
8  just_ate(stork,frog).
```

Ejercicio.: ¿Qué salida arroja Prolog si la consulta es $is\_digesting(stork, mosquito)$?

- $is\_digesting(stork, mosquito)$ unifica con la primera regla, esto quiere decir que buscará $just\_ate(stork, mosquito)$. Como esto es falso, entonces sigue evaluando las demás posibilidades.
- $is\_digesting(X, Y)$ unifica con la segunda regla. El goal por el cual Prolog reemplaza a $is\_digesting(X, Y)$ es por $just\_ate(stork, Z)$ y $is\_digesting(Z, mosquito)$.
  - $just\_ate(stork, Z)$: No hace match con $just\_ate(mosquito, blood(john))$ por falla en clash.
  - $just\_ate(stork, Z)$: no hace match con $just\_ate(frog, mosquito)$ por falla en clash.
  - $just\_ate(stork, Z)$: hace match con $just\_ate(stork, frog)$, por lo tanto el posible valor de Z es frog.
  - Por lo tanto, ahora el goal se reduce a $is\_digesting(frog, mosquito)$, y como esto es un hecho y el goal ahora está vacío, Prolog demuestra que la consulta es satisfactible y la unificación para que lo sea es Z = frog.

Haskell

Para ejecutar un archivo hay que instalar GHCI. Una vez instalado, nos paramos en la terminal en el directorio donde está el archivo que queremos ejecutar.

- Cargar archivo: :l nombreArchivo
- Ver tipo: :type tipo
- Ejecutar funcion: funcion parametro1 parametro2...
- Recargar archivo: :r
- Si necesitamos hacer cálculos para mandar un parámetro, usar paréntesis: Ej.: otherwise = n * factorial(n-1)

Maybe

El Maybe se utiliza en Haskell para recibir/devolver respuestas condicionales que pueden ser de un tipo u otro.

Se define como $data\ Maybe\ a\ =\ Nothing\ |\ Just\ a$

Ej.: $devolverFalsoSiVerdadero\ :\ Bool \rightarrow Prelude.Maybe\ Bool$

El Maybe deja la puerta abierta a un valor posible "Nothing". Entonces tenemos dos casos: Si me envian un True devuelvo False (tipo bool), caso contrario, devuelvo Nothing.

Either

El Either se utiliza en Haskell para poder recibir/devolver un parámetro que podría ser de un tipo u otro.
Se define como $data\ Either\ a\ b\ =\ Left\ a\ |\ Right\ b$

Para poder saber qué operación hacer según el tipo literalmente en código usamos (Left valor) o (Right valor).

Ej.: $devolverRepresentacionIntBool\ ::\ Either\ Int\ Bool\ \rightarrow\ Int$

Si es un entero, devuelvo ese mismo entero porque no hago nada. Eso lo hacemos con $Left(a)\ =\ a$, ahora, si el tipo es booleano tengo que decir explícitamente la respuesta según su valor. Es decir, $Right(False)\ =\ 0$ sino, $Right(True)\ =\ 1$.

Declaración de tipos en Haskell

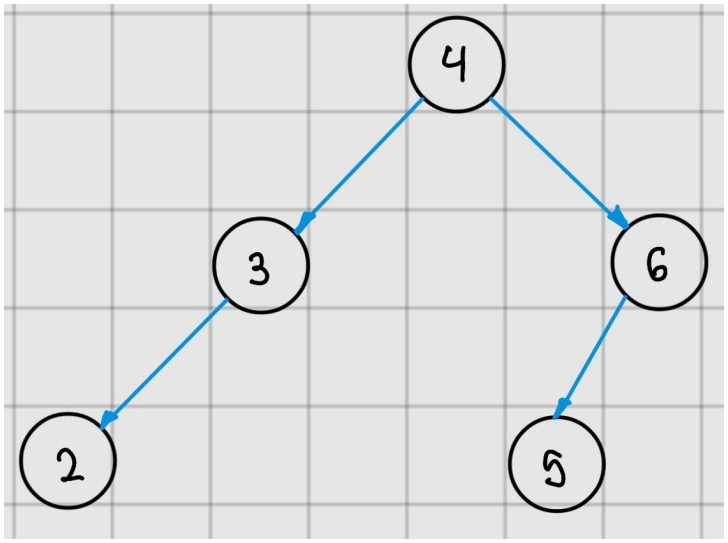
Se utiliza $data\ nombretipo\ tipo\ =\ Tipo\ 1\ |\ Tipo\ 2$ El | se interpreta como **o bien**

Árboles Binarios

Es un tipo (para mi parecer) meramente recursivo.
$data\ AB\ a\ =\ Nil\ |\ Bin\ (AB\ a)\ a\ (AB\ a)$ Nótese que es algo re contra recursivo, porque para definir el tipo de AB a decimos que es un Bin que a su vez es de AB a y a su vez AB a es otro árbol binario. Veamos unos ejemplos de esto

- Bin (Nil) Nil (Nil): es el árbol que no tiene ni siquiera raíz. Y nótese que en cada paréntesis es importante indicar el Nil pues es la forma de que el tipado de Haskell nos lo acepte.
- Bin (Bin Nil 3 Nil) 4 (Bin Nil 6 Nil): Es el árbol que comienza con un Nodo raíz que tiene el valor de 4. El hijo izquierdo del Nodo con valor 4 es otro árbol binario que tiene como valor 3 en su nodo y no tiene hijos. El hijo derecho del Nodo con valor 4 es otro árbol binario que tiene como valor 6 en su Nodo y no tiene hijos.

Y así sucesivamente, veamos un dibujo para tener algo más visual.
El siguiente árbol binario: Bin (Bin (Bin Nil 2 Nil) 3 Nil) 4 (Bin (Bin Nil 5 Nil) 6 Nil) representa el siguiente:



Curry & Uncurry

Digamos que necesitamos currificar una función que recibe una tupla de elementos. Es decir, algo así: $suma\ ::\ (Int, Int) -> Int$
Por la definición de curry necesitamos que por cada argumento, haya una función que lo devuelva, por lo tanto el resultado sería

algo así $suma :: Int -> Int -> Int$.
Veamos el tipo de función que queremos currificar: $((a, b) -> c)$, esto lo queremos llevar a $a -> b -> c$.
Por lo tanto nuestra función curry sería algo así:

```
curryOwn :: ((a, b) -> c) -> a -> b -> c
curryOwn f a b = f (a, b)
```

Entonces, digamos que queremos hacer la suma currificada.

```
sumTuple :: (Float, Float) -> Float
sumTuple (x, y) = x + y

sumarCurry :: Float -> Float -> Float
sumarCurry = curryOwn sumTuple
```

Lo que hace sumarCurry es llamar a curry(sumTuple) es decir, a curry le manda la función sumTuple. Los parámetros que le mandamos a sumarCurry como a -¿b, los convierte en (a, b) para poder aceptar el tipo de la función sumTuple.

¿Cómo sería entonces la función uncurry? Si recibimos los argumentos en forma de $a -> b -> c$ debo llevarlo a $(a, b) -> c$

```
uncurryOwn :: a -> b -> c -> ((a, b) -> c)
uncurryOwn f (a, b) = f a b

sumarUncurry :: (a, b) -> c
sumarUncurry = uncurryOwn sumarCurry
```

Esto quiere decir que vamos a llamar a sumarUncurry que recibe la tupla, ahora sumarCurry está currificada, por lo que la tenemos que convertir nuevamente a la función no currificada, para luego llamar a sumTuple de la manera original.

Clases de Tipos

- Num a: Indica que el parámetro a es numérico
- Ord a: Indica que el parámetro a es ordenable bajo algun criterio, es decir, podemos aplicar $> < =$ etc.
- Eq a: Indica que el parámetro a se puede igualar, es decir, podemos aplicar $=$

Foldr

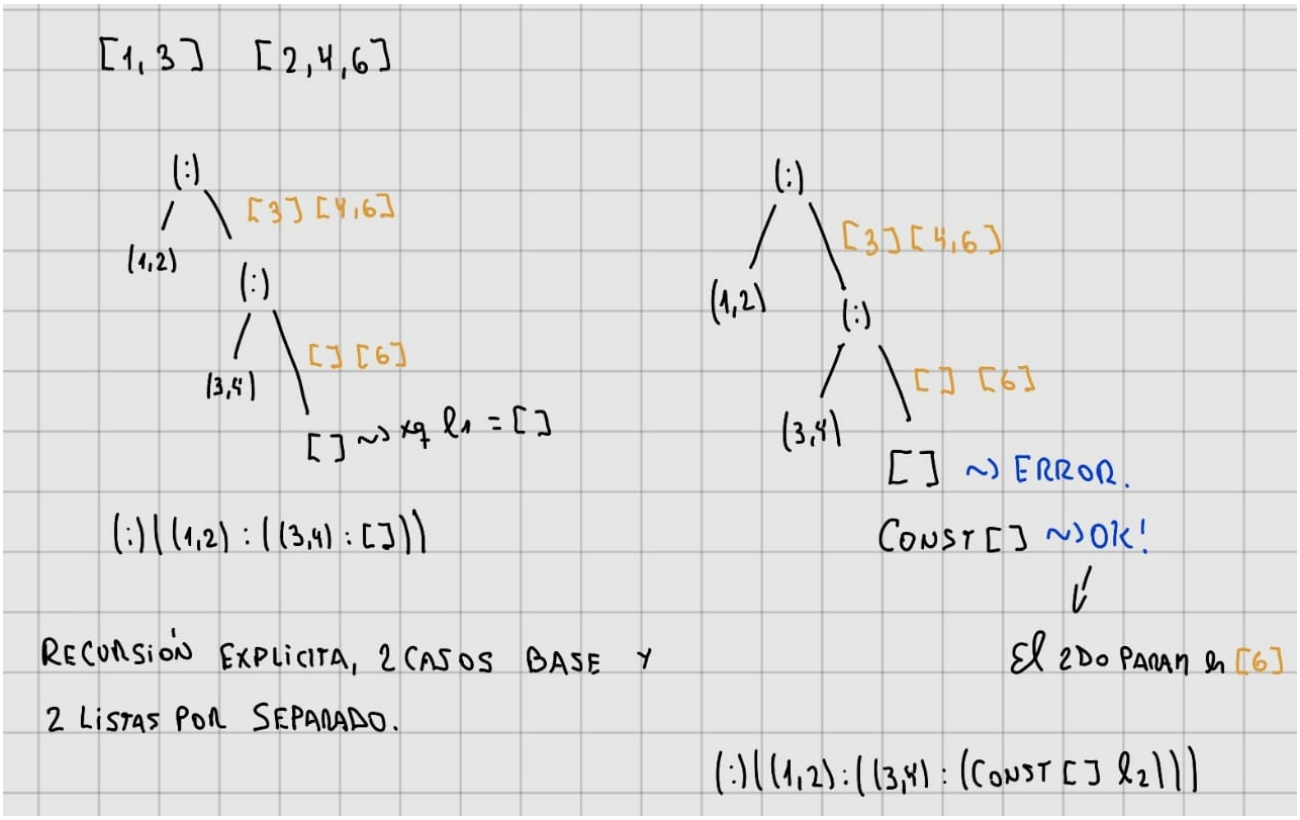
```
// Solo recorre listas de tipo a. Es decir, devuelve la suma de los elementos.
sumFoldrlist :: Num a => [a] -> a
sumFoldrlist = foldr (\x ac -> x + ac) 0

//Recorre tipos plegables. Acá no nos limitamos solo a listas, porque véase que usamos t a en vez de
    [a]
sumFoldr :: (Foldable t, Num a) => t a -> a
sumFoldr = foldr (\x ac -> x + ac) 0
```

Foldr, el árbol de recursión y más de una lista

Uno de los problemas más normales es tener que enviar más de una lista a procesar a una función dada en Haskell y realizar recursión estructural.
Veamos el siguiente ejercicio: Arme pares de la forma [(a, b)] usando recursión estructural.
Esto es súper simple si lo hacemos sin recursión estructural porque nos queda algo así

```
armarPares :: [a] -> [b] -> [(a, b)]
armarPares [] _ = []
armarPares _ [] = []
armarPares (x:xs) (y:ys) = (x, y) : armarPares xs ys
```

¿Cómo hacemos esto con foldr? Veamos el árbol recursivo.



El error en la recursión estructural vendría del lado de que como estamos recorriendo dos listas a la vez, y mi llamado recursivo es del tipo $[b] \to [(a, b)]$ no puedo devolver [] entonces lo que hago es devolver $const\ []$ y se aplica parcialmente al argumento que sería la segunda lista $const\ []\ l2$ y como la primera lista está vacía entonces devuelve [].
Esto es súper importante a tener en cuenta, porque si mandamos más de un argumento en la recursión, recordar el concepto de curry.

Flip

Toma dos parámetros y devuelve una función que los devuelve en el orden inverso.
Es decir: $(a \to b \to c) \to b \to a \to c$
Luego: $flip\ f\ a\ b\ =\ f\ b\ a$

Función identidad

Devuelve el mismo valor aplicado a la función.
Es decir: $(a \to b) \to a \to b$
Luego: $\$f\ a\ =\ f\ a$

Función constante

Devuelve un valor enviado sin aplicarle ninguna función.
Es decir: $a \to b \to a$
Luego: $const\ a\ b\ =\ a$

Reduciendo expresiones elegantemente

- $filter(\backslash x \to length\ x > 3)$
  - ¿Puede hacerse algo mejor? No. Porque a x si o sí necesitamos aplicarle una función.
- $filter(\backslash x \to x > n)$
  - ¿Puede hacerse algo mejor? Sí. $filter(> n)$
- $filter(\backslash x \to mod x 2 / = 0)$
  - ¿Puede hacerse algo mejor? No. Porque a x le tenemos que calcular su módulo con 2.

- $map(\backslash x \rightarrow map(\backslash y \rightarrow toUpper\ y)\ x)$
  - ¿Puede hacerse algo mejor? Primero entendamos que hace, recorre una lista de palabras, luego en cada palabra toma cada letra y la pasa a mayúscula. Esto es un doble map, uno por palabra otro por letra. Entonces sí $map\ (map\ toUpper)$
- $doblarElementos.filtrarPares$
  - ¿Puede hacerse algo mejor? No. Esto es el equivalente a un lenguaje imperativo hacer doblarElementos(filtrarPares(lista))

¿Qué hacen las siguientes funciones compuestas?

```
flip($) 0 id

(==0) . (flip mod 2)

Primero veamos que hace flip mod 2.
mod 2 es notación infija (Integral a => 2 -> a -> a), entonces lo que está diciendo es que si le paso cualquier número va a hacer mod 2 x, y nosotros por lo que yo entiendo es que queremos ver si es par.
Por lo tanto, lo primero que haríamos es invertir los argumentos de mod 2 con flip (Integral a => a -> 2 -> a), entonces quedaría algo como mod x 2 donde el x lo tenemos que enviar nosotros.
Luego, se compone la función de mod x 2 == 0 esperando solo un argumento donde verifica si efectivamente un número dado es par.
Entonces, (Integral a => x -> Bool)

map f = ((:) . f)
Lo que hace esta función es básicamente aplicar una función f a todos los elementos y agregarlos a una lista particular.

Dado ["hola", "abc"] quiero devolver ["cba", "aloh"]. Es decir, dar vuelta cada caracter de cada palabra y ademas dar vuelta las palabras.

reverseAnidado :: ["String"] -> ["String"]
reverseAnidado = reverse . (map . reverse)
Lo primero que hacemos es hacer un map haciendo reverse por cada caracter de la lista. Luego, reordenamos las palabras en sí.
El tipo de reverse es: [a] -> [a] pero con los elementos al revés. Entonces, por cada palabra (map) hacemos un reverse y las guardamos.
Finalmente, nos queda algo así ["aloh", "cba"], nos queda dar vuelta eso, entonces hacemos nuevamente un reverse de toda la lista. ["cba", "aloh"].

listacomp f xs p = [f x | x <- xs, p x]
listacomp f xs p = map f (filter p xs)
```

Ejercicios Foldl

1. Definir la función sumasParciales que dada una lista de números devuelve otra de la misma longitud que tiene en cada posicion la suma parcial de los elementos de la lista original desde la cabeza hasta la posición actual.
Entendamos el enunciado:

- Vamos a usar foldl para ir sumando de izquierda a derecha.
- El tipado de foldl es b → a → b donde b es nuestro primer argumento acumulador y a el elemento.
- Necesito de alguna manera tener el valor inmediato anterior. Podemos hacer algo como empezar enviando una lista vacia como caso base, y a medida que vamos haciendo la recursion tomar la cabeza de la lista.
- Si la lista esta vacia entonces solo agrego x a la lista de la recursion (primer elemento), si no esta vacia sumo con el elemento de la lista (cabeza)
- Porque foldl labura asi → [1, 2, 3]
  - 1:[] = [1]
  - 2 + (head [1]) : [1] = [3, 1]
  - 3 + (head [3, 1]) : [3, 1] = [6, 3, 1]

- Ahora podemos usar reverse y el resultado es [1, 3, 6]

Entonces la solución sería algo así:

```
sumasParciales :: Num a => [a] -> [a]
sumasParciales = reverse . foldl(\acc x -> if(length acc > 0) then x+(head acc):acc else x:acc) []
```

Ejercicios Map

1. Realice una función mapDoble que toma una función currificada de dos argumentos y dos listas de igual longitud y devuelve una lista de aplicaciones de la función a cada elemento correspondiente de las dos listas.
Básicamente $f = x + y$ $l1 = [1, 2]$ $l2 = [3, 4]$ da como resultado $[4, 6]$
Desglosemos el ejercicio en partes

- 1. Lo primero que necesito hacer básicamente es recorrer ambas listas de alguna manera a la vez, y obtener algo como [(1, 3), (2, 4)] y luego aplicar a esa lista de pares la función f. Si nos ponemos a pensar, basta con hacer una función que reciba dos listas de tipo [a] y [b] y devuelva una lista de [(a, b)].
- 2. Una vez que tenemos esta lista de pares, sabemos que la función que nos va a enviar tiene que utilizar ambos elementos a la vez, pero la función es de la forma $a \to b \to c$ y esto quiere decir que está currificada pero nuestra lista de pares es de tipo $[(a, b)]$ por lo tanto antes de aplicar f debemos aplicar $uncurry\ f\ lista$ para que cuando mandemos f y la lista, f se convierta en una función que espere (a, b).
- 3. Por último, para aplicar a todos los elementos de la lista podemos usar map de la siguiente forma: $map\ (uncurry f)(lista)$

```
mapDobleCorta :: (a -> b -> c) -> [a] -> [b] -> [c]
mapDobleCorta f l r = map (uncurry f) (armarPares l r)
```

donde la función armarPares tiene la siguiente pinta

```
armarPares :: [a] -> [b] -> [(a, b)]
armarPares _ [] = []
armarPares [] _ = []
armarPares (x:xs) (y:ys) = (x, y) : armarPares xs ys
```

2. Realice una suma de matrices.

- Idea: Necesitamos de alguna manera recorrer la fila 1 de la matriz 1 y la fila 1 de la matriz 2. Esto lo podemos hacer facilmente reutilizando el ejercicio anterior (armarPares), es decir, enviamos armarPares con la fila1 matriz1 y fila1 matriz2. Esto nos armaría los pares de esa fila.
- Tenemos que generalizar este proceso para cada fila. Por lo tanto podemos recibir una matriz, y podemos hacer recursion sobre cada lista de la matriz.
- A su vez, vamos a necesitar que una vez que tenemos los pares armados, sobre esos pares se aplique un map haciendo uncurry sobre f. Porque si tenemos F1 M1 + F1 M2 = [(1, 2), (3, 4), (5, 6)] esto indica que la fila de la M1 es [1, 3, 5] y la fila de la M2 es [2, 4, 6]. Por lo tanto, lo que necesito hacer es convertirlo en [[9, 12]] y agregarlo a la lista resultante. El Uncurry acá es ultra importante porque mi función pide $Int \to Int \to Int$ y yo voy a mandar $(Int, Int) \to Int$.

```
sumaMat :: (Int -> Int -> Int) -> [[Int]] -> [[Int]] -> [[Int]]
sumaMat _ [] _ = []
sumaMat _ _ [] = []
sumaMat f (x:xs) (y:ys) = map (uncurry f) (armarPares x y) : sumaMat f xs ys
```

¿Qué es lo que podríamos cambiar? Estamos haciendo un laburo exactamente igual mapDoble.

```
sumaMat :: (Int -> Int -> Int) -> [[Int]] -> [[Int]] -> [[Int]]
sumaMat f = mapDoble(mapDoble(f))
```

Armando funciones que permitan hacer recursión sobre un tipo dado

1. **foldNat**: Necesitamos hacer recursión sobre los números enteros. Una excelente pregunta es ¿recursión sobre números naturales?. Sí.
Un número natural se define de la siguiente manera $data\ Nat\ =\ Zero \mid Succ\ Nat$. Es decir, tiene dos chances: O es cero, o es un sucesor de algún número.
¿Cuántos casos tendríamos que probar si quisieramos verificar la correctitud del tipo? 2. Que sea Zero o que sea algún sucesor.
Así, de esta manera, podemos definir al número 4 como $Succ(Succ(Succ(Succ\ Zero)))$.

La recursión nos sirve justamente para esto, para poder hacer operaciones con números naturales.
Ej.: Necesitamos multiplicar un número n m veces ¿Cómo hacemos esto? Sumamos el mismo número m veces. Es decir, si quiero hacer n * n equivale a decir n+n+n+n+n. Entonces ¿Cómo podríamos hacer esto?
Para empezar, pensemos en qué tipo de operaciones queremos hacer con foldNat. Podríamos hacer multiplicación, potencia, etc.
Pensemos por un momento ¿qué pasaría si el caso base fuese 0 si estamos sumando? Nada, porque justamente para la multiplicación la queremos hacer como n+n+n+n+n y si llegamos a Zero quiero que devuelva 0.
Ahora ¿qué sucede si queremos hacer la potencia? Recordemos que la potencia se define como la multiplicación de un número m veces. Si nos abstraemos a nuestro esquema $n^m \equiv n * n * n * n...m \equiv (n+n)+(n+n)+(n+n)+(n+n)...m\ veces$ si quisieramos aplicar el caso base de 0 para la multiplicación se nos haría 0. Por lo tanto tenemos otro caso base, sería 1.
Por lo tanto, definamos el foldNat pero utilizando el tipo de Integer (como pidió la cátedra)

```
1    foldNat :: Integer -> (Integer -> Integer) -> Integer -> Integer
2    foldNat base _ Zero = base
3    foldNat base f n = f (foldNat base f (n-1))
```

Entonces ahora podemos definir la multiplicación como

```
1    multiplicacion :: Integer -> Integer -> Integer
2    multiplicacion n m = foldNat 0 (+n) m
```

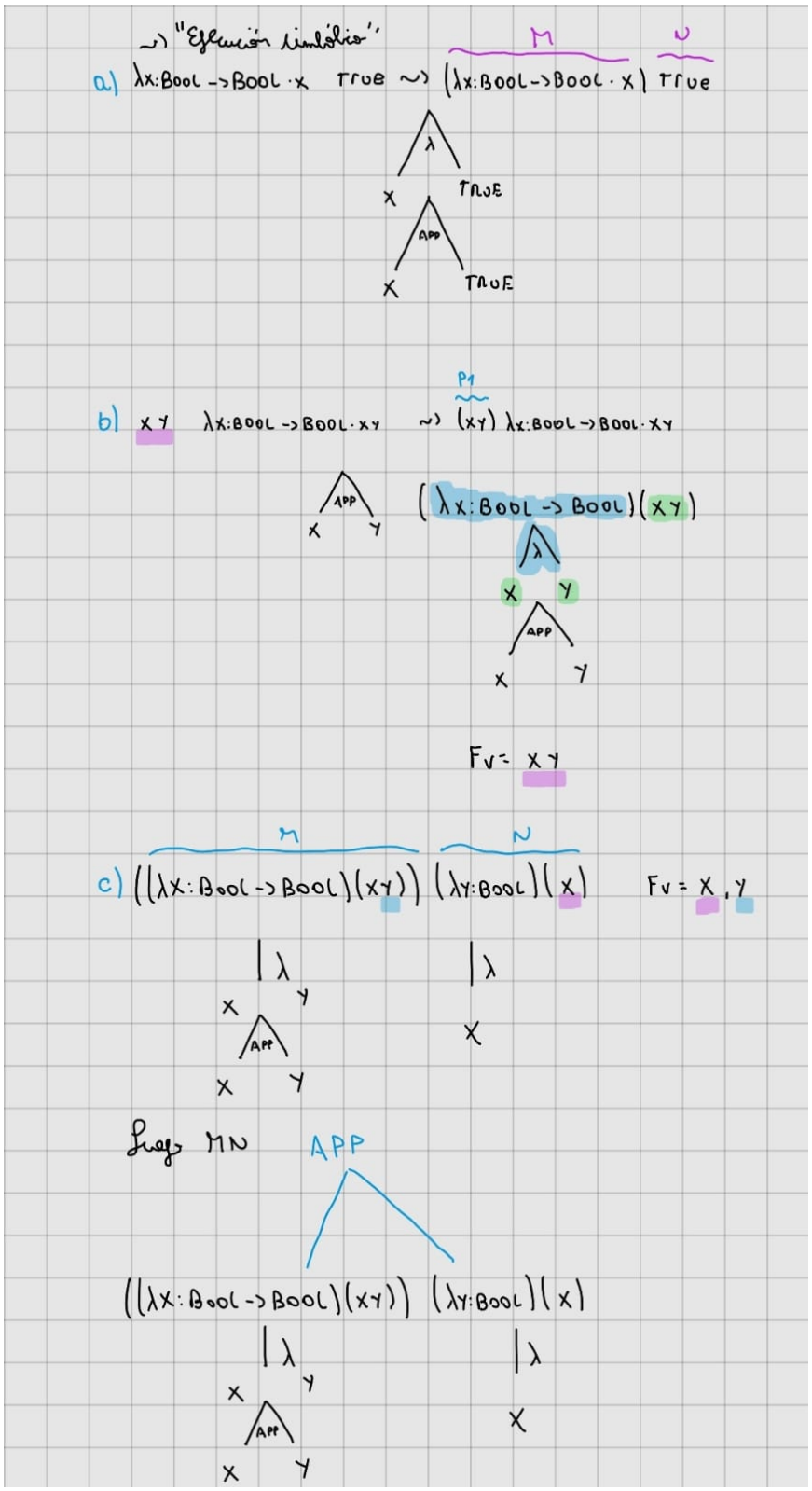
¡Nótese que acá el caso base es 0 porque estamos sumando!
Por último podemos definir la potencia reutilizando la multiplicación

```
1    potencia :: Integer -> Integer -> Integer
2    potencia n m = foldNat 1 (multiplicacion n m) m
```

Es importante notar que la potencia requiere hacer foldNat nuevamente porque tenemos que hacer el proceso de multiplicación m veces.

¿Son términos válidos?

Recordemos que asocia a la derecha las implicaciones y hacia izquierda la aplicación.
Lo que hay que revisar bien siempre son todos los tipos.



La idea es ir aplicando como Haskell la asociación, separar en varios pasos la ejecución y luego aplicarlo.

Términos LPO

Sea $=\{d, f, g\}$ donde d tiene aridad 0, f aridad 2 y g aridad 3. ¿Cuales de la siguientes cadenas son términos sobre $\mathcal{F}$?
Recordemos definiciones:

- Si un Símbolo de Función tiene aridad 0: es constante.

- Si f tiene una aridad n, cuando se utilice deben enviarse los n parámetros.
- Un término tiene la pinta: $t ::= X \mid f(t_1, \ldots, t_n)$

1. g(d,d): No es término. g es un símbolo de función pero de aridad 3 y acá se le están enviando solo dos.
2. f(X, g(Y, Z), d): No es término. Mismo caso que arriba, f está mal aplicado y g también. Aridades incorrectas.
3. g(X, f(d, Z), d): Es un término. f y g están aplicados con su aridad esperada, y los elementos que se envían por parámetro son términos irreducibles y/o constantes.
4. g(X, h(Y, Z), d): h no está definido en nuestro conjunto de Símbolos de Funciones.

Fórmulas Válidas, uso de Predicados y Funciones

Sea c una constante, f un símbolo de función de aridad 1 y S y B dos símbolos de predicados binarios. ¿Cuales de las siguientes son fórmulas?
Recordemos la teoría

- Una constante C es una función con aridad 0.
- Las funciones o símbolo de función esperan siempre los n argumentos con los cuales se definieron. Devuelven un elemento del dominio.
- Los predicados S y B, binarios (reciben dos argumentos) hacen una relación específica entre dos términos irreducibles y arrojan un valor de verdad.

1. S(c, X): Es una fórmula, estamos comparando una constante c con un valor irreducible X. Devuelve un valor de verdad.
2. B(c, f(c)): Es una fórmula, estamos comparando una constante c con un valor irreducible que produce f(c).
3. f(c): No es una fórmula. Es un valor del dominio, no reduce a un valor de verdad.
4. B(B(c, X), Y): No es una fórmula, un predicado no puede recibir como parámetro un valor de verdad (B(c, X)).
5. S(B(c), Z): Mismo caso que el anterior
6. $(B(X,Y) \implies (\exists Z.S(Z,Y)))$: Es una fórmula. B(X,Y) es valor de verdad, $\implies$ define una fórmula y $(\exists Z.S(Z,Y))$ es una fórmula pues el Z que existe está en nuestro dominio, y $S(Z,Y)$ relaciona dos elementos de nuestro dominio y arroja un valor de verdad.
7. $(S(X,Y) \implies S(Y,f(f(X))))$: Es una fórmula. S(X,Y) es una fórmula, $\implies$ es una fórmula, y $S(Y,f(f(X)))$ es una fórmula porque al aplicar dos veces f, nos termina devolviendo un elemento de nuestro dominio que luego es comparado con Y en el símbolo de predicado S.
8. $B(X,Y) \implies f(X)$: No es una fórmula. f(X) es un término o valor de nuestro dominio, no es un valor de verdad. Se le debería aplicar un símbolo de predicado para que arroje un valor de verdad.
9. $S(X,f(Y)) \land B(X,Y)$: Es una fórmula. 10. $\forall X.B(X,f(X))$: Es una fórmula. Para todo elemento posible de nuestro dominio X, al enviarlo a f nos devuelve un valor del dominio y se lo compara con el X. Luego, aplicar B devuelve un valor de verdad.
11. $\exists X.B(Y,X(C))$: No es una fórmula, porque X es un elemento del dominio y acá se lo está usando como función de aridad 1 (creo)

Pasaje de LPO a Forma Clausal

1. Sea $\sigma = \exists X.\forall Y.(P(X,Y) \land Q(X) \land \neg R(Y))$

- Reemplazamos todas las ocurrencias de implicaciones por su equivalente. Como acá no hay, no hacemos nada.
- Reemplazamos todas las ocurrencias de las negaciones exteriores hacia adentro, como acá están todas dentro, no hacemos nada.
- Movemos todos los cuantificadores hacia afuera. Como acá no hay adentro, no hacemos nada.
- Pasamos los cuantificadores existenciales a función o constante, en este caso como el existencial está fuera de todo (no depende de ningun cuantificador universal), entonces nuestra hipotética x, será una constante. Por lo tanto $= \forall Y.(P(C,Y) \land Q(C) \land \neg R(Y))$
- Distribuimos los $\lor$ sobre $\land$. Como acá no hay, no hacemos nada.
- Por último, los cuantificadores universales por cada $\land$ distribuimos. Por lo tanto $= \forall Y.P(C,Y) \land \forall Y.Q(C) \land \forall Y.\neg R(Y)$
- Entonces la forma clausal es: $\mathcal{C} = \{\{P(C,Y)\}, \{Q(C)\}, \{\neg R(Y)\}\}$