DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

Programación Funcional

2 de Abril de 2025 — Paradigmas de Lenguajes de Programación

Integrante	LU	Correo electrónico
Solana Navarro	906/22	solanan3@gmail.com
Melli, Tomás Felipe	371/2	tomas.melli1@gmail.com
Lourdes Wittmund Montero	1103/22	lourdesmonterochiara@gmail.com
Marco Romano Fina	1712/21	marcoromanofinaa@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1 Introducción

Para este trabajo práctico vamos a implementar dos módulos.

1. **Documento** : donde definiremos un tipo de dato Doc y funciones para trabajar con docu- mentos. Un documento es una estructura cuyo objetivo es mostrarse por pantalla en forma amigable para el usuario.
2. **PPON** : donde definiremos un tipo de dato PPON y funciones para trabajar con este formato de datos compuesto. Además, definiremos cómo convertir un PPON a un Doc para mostrarlo en forma amigable.

2 Módulo Documento

Un documento está representado por la estructura recursiva Doc con la siguiente definición:

```
data Doc = Vacio | Texto String Doc | Linea Int Doc
```

- Vacio: representa un documento vacıo.
- Texto: representa un documento que contiene como primer componente un texto.
- Linea: representa un documento que contiene como primer componente un salto de linea. La siguiente linea comienza con una cantidad de espacios indicada por un entero.

Un valor del tipo Doc debe cumplir con los siguientes invariantes: Sea Texto s d entonces:

- s no debe ser el string vacıo.
- s no debe contener saltos de lınea
- d debe ser Vacio o Linea i d'

Sea Linea i d entonces:

- $i \geq 0$

Ya tenemos definidas las funciones : *vacío, linea y texto.*

2.1 Qué es Doc ?

Doc es un tipo de dato algebraico recursivo(esto es porque alguno de sus constructores refiere a sí mismo). Tenemos **tres** maneras de construir datos de este tipo, miremos sus constructores :

1. **Vacio** : crea un documento vacío.
2. **Texto String Doc** :crea un documento que tiene texto (el string del cual se compone) y otro **documento**, este es aquel que le sigue. Por ejemplo:

```
ghci> miText1 = texto "Hola"
ghci> miText2 = texto "Mundo"
ghci> miText1
ghci> Texto "Hola" Vacio
ghci> miText2
Texto "Mundo"
```

3. **Linea Int Doc**: crea un documento que tiene una línea numerada, y otro documento que le sigue en la misma línea. Ejemplo :

```
ghci> miTextLinea = linea
ghci> miTextLinea
ghci> Linea 0 Vacio
```

2.2 Ejercicio 1

En este ejercicio tenemos que crear un **foldDoc**, es decir, una abstracción de un esquema de recursión estructural para recorrer los documentos. Como vimos en la práctica, seguimos la receta :

```
foldDoc :: b -> (String -> b -> b) -> (Int -> b -> b) -> Doc -> b
foldDoc fVacio fTexto fLinea doc =
  case doc of
  Vacio -> fVacio
  Texto s d -> fTexto s (foldDoc fVacio fTexto fLinea d)
  Linea i d' -> fLinea i (foldDoc fVacio fTexto fLinea d')
```

Veamos por partes :

- **fVacio** aplica fVacio. Tipo b
- **fTexto** que recibe un String un doc s y el documento foldeado. Tipo (String $\rightarrow$ b $\rightarrow$ b)
- **fLinea** recibe una línea y el documento foldeado. (Int $\rightarrow$ b $\rightarrow$ b)

2.3 Ejercicio 2

En este ejercicio queremos definir $< + >$, el operador que **concatena documentos**. Tenemos 3 aserciones que debemos usar para chequear que anda bien, y el invariante. Este último nos dice : Sea Texto s d:

- s no debe ser el string vacıo.
- s no debe contener saltos de lınea
- d debe ser Vacio o Linea i d'

Sea Linea i d entonces:

- $i \geq 0$

Tenemos casos :

1. si d1 es un doc vacío entonces devolvemos d2.
2. si d1 no es vació entonces tenemos casos
   - si d1 termina en linea, lo dejo como está
   - si d1 termina en texto y d2 tiene texto entonces, Text (d1.text (++) d2.text) el resto del doc

Definimos dos auxiliares para que nos ayude a capturar : que es el final de **d1** y para unir los textos :

```
final :: Doc -> Bool
final (Texto s d) = if d == Vacio then True else False

unirTextos :: String -> Doc -> Doc
unirTextos s (Texto s' d) = Texto (s++s') d
```

Con esto en mente, usamos **foldDoc** para implementar el operador:

```
(<+>) :: Doc -> Doc -> Doc
(<+>) d1 d2 = foldDoc
    d2
    (\text rec -> if final rec then unirTextos text rec else Texto text rec)
    (\line rec -> Linea line rec)
    d1
```

Veamos por partes :

1. En caso de que d1 sea **vacío**, fVacío evalúa a d2.
2. Para el caso en que estemos presenciando **texto**, tenemos que chequear que sea el final de d1. Con la auxiliar sacamos esa info. En caso de ser el final, con la auxiliar concatenamos los textos y construimos el nuevo doc. Caso contrario, construimos el texto con el doc que estaba.
3. En el caso de estar frente a una **línea**, simplemente construimos la línea con el documento que tenía.

4. Le pasamos como documento a **foldear** a d1

Con respecto al invariante:
En el caso fTexto, estamos creado el documento a partir del constructor y por tanto, s no es el string vacío, no tiene saltos de línea y d es rec (que si es Vacío, vale; sino es Línea con lo cual Texto text rec representa ese caso).
Para el caso de fLinea, es análogo, usamos el constructor sin modificar.
Como devolvemos un documuento bien formado en caso de ser d1 Vacío, también vale.

2.4 Ejercicio 3

En esta parte del módulo tenemos que implementar la función **indentar** que logra agregar cierta cantidad de espacios al principio de cada línea del documento. **No modifica la primera, sólo las siguientes a la primera**. La idea es, si el documento está vacío simplemente, evaluamos a vacío. Si el documento tiene formato texto, lo dejamos tal cual, pero si vemos que el documento es de "tipo" línea, vamos a querer aplicarle la indentación.

```
indentar :: Int -> Doc -> Doc
indentar i = if i < 0 then error "No se puede indentar negativo" else foldDoc
  Vacio
  (\text rec -> Texto text rec)
  (\line rec -> Linea (line + i) rec)
```

Vale el invariante pues :
Sea el documento *Texto s d*, como lo creamos a partir de su constructor que cumple con la no-presencia del salto de línea ni el string vacío. Vale el invariante
Sea el documento *Linea i d*, cumple el invariante ya que $\forall i\ (line + i) \geq 0$

2.5 Ejercicio 4

La función **mostrar** convierte el documento en un string listo para mostrar por pantalla. Para el caso del documento vacío, lo mostramos como **" "**. Para un documento de texto, lo mostramos como tal, pero hay un detalle para los documentos que están separados por líneas. Para ellos, tenemos que modelar el comportamiento tal cual lo vemos en la realidad : un salto de línea y si está indentado, replicar los espacios de indentación. La función **replicate** que tiene la signatura :

```
replicate :: Int -> a -> [a]
```

Nos permite como su nombre indica, replicar cierta cantidad de veces un dato, en este caso, nos interesan los espacios. Entonces, *line-veces* **' '**.

```
mostrar :: Doc -> String
mostrar = foldDoc
  ""
  (\text rec -> text ++ rec)
  (\line rec -> "\n" ++ replicate line ' ' ++ rec)
```

3 Módulo PPON

Este módulo importa de Doc : module Documento (Doc, vacio, linea, texto, foldDoc,(¡+¿), indentar, mostrar, imprimir,) where. Pero detalle : **Doc** está solo, es decir, **oculta la implementación interna del tipo**. No podremos inspeccionar ni construir valores de tipo Doc directamente (no se puede pattern-matching). No podemos hacerlo, pero si modificamos en el módulo doc (Doc(...), ..) le damos acceso.

3.1 Qué es PPON?

PPON es un tipo de dato algebraico recursivo también, ya que alguno de sus constructores hace referencia a sí mismo (a su mismo tipo).

```
data PPON
  = TextoPP String
  | IntPP Int
  | ObjetoPP [(String, PPON)]
  deriving (Eq, Show)
```

1. **TextoPP** es el nombre del constructor del tipo en el que se almacena *texto*

2. **IntPP** es el nombre del constructor del tipo en el que se almacena un *entero*
3. **ObjetoPP** es el nombre del constructor del tipo que se define recursivamente en el que se almacena una estructura de la forma *[(String, PPON)]*. Podemos pensarlo como un diccionario, un array de claves-valor.

Nos dan unos ejemplos :

- pericles = ObjetoPP [("nombre", TextoPP "Pericles"), ("edad", IntPP 30)]
- merlina = ObjetoPP [("nombre", TextoPP "Merlina"), ("edad", IntPP 24)]
- addams = ObjetoPP [("0", pericles), ("1", merlina)]

Ejemplos de creación :

```
ghci> ppon1 = TextoPP "Soy un ppon atomico"
ghci> ppon2 = IntPP 1000
ghci> ppon3 = ObjetoPP [("Tomas", TextoPP "Melli")]
```

3.2 Ejercicio 5

Tenemos que definir un **predicado** para determinar si un PPON es **atómico** (si es TextoPP o IntPP).

```
pponAtomico :: PPON -> Bool
pponAtomico (TextoPP str) = True
pponAtomico (IntPP i) =  True
pponAtomico _ = False
```

Con **foldPPON** :

```
pponAtomico :: PPON -> Bool
pponAtomico = foldPPON (const True) (const True) (const False)
```

3.3 Ejercicio 6

Tenemos que volver a definir un **predicado**, pero en este caso, tenemos que determinar si un PPON es un **objeto con valores atómicos**. Recién cuando presentamos el constructor recursivo del tipo dijimos que podemos abstraer la estructura a la de un diccionario. Con ese modelo en mente, esta función debería decirnos si el valor de cierta clave es un pponAtómico.
Una idea fue pensar en hacer un **foldPPON** para foldear el tipo:

3.3.1 foldPPON

```
foldPPON :: (String -> b) -> (Int -> b) -> ([(String, b)] -> b) -> PPON -> b
foldPPON fTexto fInt fObjeto ppon =
  case ppon of
    TextoPP str -> fTexto str
    IntPP i -> fInt i
    ObjetoPP [(str, p)] -> fObjeto str (foldPPON fTexto fInt fObjeto p)
```

Esto parecería tener sentido... pero el problema es que, la última línea matchea si el PPON *es una lista con un sólo par (clave-valor)*. Entonces, por eso hay que redefinir el esquema de *pattern matching* para que matchee correctamente :

```
foldPPON :: (String -> b) -> (Int -> b) -> ([(String, b)] -> b) -> PPON -> b
foldPPON fTexto fInt fObjeto ppon =
  case ppon of
    TextoPP str -> fTexto str
    IntPP i -> fInt i
    ObjetoPP pares ->
      let pares' = [(clave , rec valor) | (clave,valor) <- pares]
      in fObjeto pares'
      where rec = foldPPON fTexto fInt fObjeto
```

Qué hace esto entonces ?
Básicamente, como el patrón que queremos matchear es un ObjetoPP, que está definido como una lista de clave-valor, tenemos que capturar esa lista, que llamamos **pares** y definir por *comprensión* la lista **pares'** que es el resultado de **recorrer cada par clave-valor del ObjetoPP y aplicarle el foldPPON**. O sea que ya tenemos, post fold, algo de este estilo [(String, b)]
Cómo probamos que funciona ?
Miremos cómo podemos extraer los textos de los ObjetoPP.

```
pericles = ObjetoPP [("nombre", TextoPP "Pericles"), ("edad", IntPP 30)]
merlina = ObjetoPP [("nombre", TextoPP "Merlina"), ("edad", IntPP 24)]
addams = ObjetoPP [("padre", pericles), ("madre", merlina)]
textosPPON :: PPON -> [String]
textosPPON = foldPPON (\s -> [s]) (const []) (concatMap snd)
----------------
ghci> textosPPON addams
["Pericles","Merlina"]
```

Sol propuso una manera copada de hacerlo :

```
-- versi n Sol :
ObjetoPP pares -> fObjeto (map (\(x, y) -> (x, rec y)) pares)
where rec = foldPPON fTexto fInt fObjeto
```

Volvamos a los que nos compete : saber si **la lista de pares clave-valor contienen valores de tipo atómico**.

```
pponObjetoSimple :: PPON -> Bool
pponObjetoSimple ppon = case ppon of
  TextoPP t -> False
  IntPP i -> False
  ObjetoPP objeto -> foldr (\(x, y) rec -> pponAtomico y && rec) True obj
```

3.4 Ejercicio 7

En esta función tenemos que intercalar un documento entre una lista de otros documentos. El ejemplo es :

```
mostrar (intercalar (texto ", ") [texto "a", texto "b", texto "c"])    "a, b, c"
```

Como no tenemos los constructores en este módulo, ya que Doc no los exporta, no podemos hacer *pattern-matching*.

```
PPON.hs:38:13: error: Not in scope: data constructor    Texto
   |
38 | intercalar (Texto s) (x:[]) = s
   |
```

Dicho esto ... podemos usar $< + >$

```
ejemploIntercalar  = intercalar (texto ", ") [texto "a", texto "b", texto "c"]

intercalar :: Doc -> [Doc] -> Doc
intercalar d (x:[]) = x
intercalar d (x:xs) =  x <+> d <+> intercalar d xs
----------------
ghci> mostrar ejemploIntercalar
"a, b, c"
```

Ahora podemos usar la función **entreLlaves**

```
ghci> imprimir (entreLlaves ejEntreLlaves )
{
  a,
  b,
  c
}
```

Acabo de leer que no vale **recursión explícita**, entonces :

```
intercalar :: Doc -> [Doc] -> Doc
intercalar d = foldr1 (\x rec -> x <+> d <+> rec)
```

Usamos **foldr1** para que no nos quede colgando ese $< + > Vacio$ que hace que quede *'últimimo-elemento, '*, ya que *foldr1* toma el caso base como el último elemento :)
Pero el test explicita un caso vacío, entonces ... atamos con alambre el caso molesto

```
intercalar :: Doc -> [Doc] -> Doc
intercalar d [] = vacio
intercalar d lista = foldr1 (\x rec -> x <+> d <+> rec) lista
```

3.5 Ejercicio 8

La función **aplanar** básicamente reemplaza los saltos de línea y su indentación con un *salto de línea*.
El ejemplo que nos dan :

```
ghci> imprimir (aplanar (texto "a" <+> linea <+> texto "b"))
a b
ghci> imprimir (aplanar (texto "a" <+> indentar 2 (linea <+> texto "b")))
a b
```

Con esto en mente...

```
ejAplanar = texto "a" <+> linea <+> texto "b"
ejAplanar' = texto "a" <+> indentar 2 (linea <+> texto "b")
aplanar :: Doc -> Doc
aplanar = foldDoc
  vacio
  (\text rec -> texto text <+> rec)
  (\line rec -> texto " " <+> rec)
```

Definimos los ejemplos, y luego procedemos a declarar la función. Como foldDoc se puede utilizar, definimos una λ para procesar el texto que simplemente se concatena y para las líneas otra λ que concatene " " a la recursión.

```
ghci> imprimir (aplanar ejAplanar)
a b
ghci> imprimir (aplanar ejAplanar')
a b
```

3.6 Ejercicio 9

En este ejercicio queremos hacer una "conversión" de un PPON a un Doc que lo represente. Nos piden cumplir :

1. Si PPON es **TextoPP**, el documento debe evaluar a **ese texto entre comillas dobles**
2. Si PPON es **IntPP**, el documento debe evaluar al **número que represente**
3. Si PPON es **ObjetoPP**, el documento debe evaluar al objeto expresado como {"clave": valor1, "clave": valor2 } donde **las claves van entre comillas dobles, y cada clave valor va en una línea diferente**.

La idea después es poder imprimirlo por pantalla como se ve en este ejemplo :

```
ghci> imprimir (pponADoc addams)
{
    "0": { "nombre": "Pericles", "edad": 30 },
    "1": { "nombre": "Merlina", "edad": 24 }
}
```

La idea es foldear un tipo PPON dando como fTexto y fInt como dice la consigna. Pero el caso se pone áspero cuando tenemos un objeto no-simple. Cosas a tener en cuenta : cuando recibimos la lista de [(string, PPON)] tenemos como "muchos documentos". La idea es que esos muchos que son "[Doc]" terminen siendo Doc a secas. Por eso, le aplicamos una función a cada tupla que los concatene $< + >$ y ahora ya son [Doc] de verdad, que si **aplanamos** cada uno, los objetos no-simples se condensan, si no, tenemos los objetos simples. Finalmente, utilizamos la función dada **entrellaves** que toma cada uno de esos [Doc] y los devuelve como Doc impresos por pantalla.

```
pponADoc :: PPON -> Doc
pponADoc (TextoPP s) = texto (show s)
pponADoc (IntPP i) = texto (show i)
pponADoc (ObjetoPP pares) =
  if pponObjetoSimple (ObjetoPP pares)
    then aplanar (entreLlaves (map (\(x,y) -> texto (show x) <+> texto ": " <+> pponADoc y) pares))
    else entreLlaves (map (\(x,y) -> texto (show x) <+> texto ": " <+> pponADoc y) pares)
```

4 Demostración

4.1 Ejercicio 10

Demostrar, utilizando razonamiento ecuacional e inducción estructural, que para todo $n, m ::$ `Int` positivos y $x ::$ `Doc` se cumple:

$$\texttt{indentar}\ n(\texttt{indentar}\ m\ x) = \texttt{indentar}\ (n+m)\ x$$

Se sugiere demostrar los siguientes lemas :

1. $indentar\ k\ Vacio = Vacio\ \forall k :: Int\ positivo$
2. $indentar\ k\ (Texto\ s\ d) = Texto\ s\ (indentar\ k\ d)\ \forall k :: Int\ positivo, s :: String\ y\ d ::: Doc$
3. $indentar\ k\ (Linea\ k\ d) = Linea\ (m+k)\ (indentar\ m\ d))\ \forall m, k :: Int\ positivo\ y\ d ::: Doc$

Tenemos las siguientes funciones con sus etiquetas para realizar la correcta demostración de los lemas y, consecuentemente, el ejercicio.

$vacio :: Doc$
$vacio = Vacio\ \{V_0\}$

$linea :: Doc$
$linea = Linea\ 0\ Vacio\ \{L_0\}$

$texto :: String -> Doc$
$textot|''`elem`t = error""\{T_1\}$
$texto\ [\,] = Vacio\ \{T_2\}$
$texto\ t = Texto\ t\ Vacio\ \{T_3\}$

$foldDoc :: b -> (String -> b -> b) -> (Int -> b -> b) -> Doc -> b$
$foldDoc\ fVacio\ fTexto\ fLinea\ doc = case\ doc\ of$
$Vacio -> fVacio\ \{FD_1\}$
$Texto\ s\ d -> fTexto\ s\ (rec'\ d)\ \{FD_2\}$
$Linea\ i\ d' -> fLinea\ i\ (rec'\ d')\ \{FD_3\}$
$where\ rec = foldDoc\ fVacio\ fTexto\ fLinea\ \{FD_4\}$

$indentar :: Int -> Doc -> Doc$
$indentar\ i = foldDoc$
$vacio\ \{I_1\}$
$(text\ rec -> Texto\ text\ rec)\ \{I_2\}$
$(line\ rec -> Linea\ (line+i)\ rec)\ \{I_3\}$

4.1.1 Probamos Lema 1

$$indentar\ k\ Vacio = Vacio\ \forall k :: Int\ positivo$$

$\overset{ppio.\ de\ reemplazo}{=} foldDoc\ vacio\ (text\ rec -> Texto\ text\ rec)\ (line\ rec -> Linea\ (line+k)\ rec))\ Vacio = Vacio$
$\overset{FD_1}{=} vacio = Vacio$
$\overset{V_0}{=} Vacio = Vacio$
Como se quería probar.

4.1.2 Probamos Lema 2

$$indentar\ k\ (Texto\ s\ d) = Texto\ s\ (indentar\ k\ d)\ \forall k :: Int\ positivo, s :: String\ y\ d ::: Doc$$

$\overset{ppio\ de\ reemplazo}{=} foldDoc\ vacio\ (text\ rec -> Texto\ text\ rec)\ (line\ rec -> Linea\ (line+k)\ rec))\ (Texto\ s\ d)$
$\overset{FD_2}{=} (text\ rec -> Texto\ text\ rec)\ s\ (foldDoc\ vacio\ (text\ rec -> Texto\ text\ rec)\ (line\ rec -> Linea\ (line+k)\ rec)\ d)$
$\overset{\beta\ y\ ppio.\ de\ reemplazo}{=} (rec -> Texto\ s\ rec)\ (indentar\ k\ d)$
$\overset{\beta}{=} Texto\ s\ (indentar\ k\ d)$
Como se quería probar.

4.1.3 Probamos Lema 3

$$indentar\ m\ (Linea\ k\ d) = Linea\ (m+k)\ (indentar\ m\ d))\ \forall m, k :: Int\ positivo\ y\ d ::: Doc$$

$\overset{ppio.\ de\ reemplazo}{=} foldDoc\ vacio\ (text\ rec- > Texto\ text\ rec)\ (line\ rec- > Linea\ (line+m)\ rec)\ (Linea\ k\ d)$
$\overset{FD_3}{=} (line\ rec- > Linea\ (line+m)\ rec)\ k\ (foldDoc\ vacio\ (text\ rec- > Texto\ text\ rec)\ (line\ rec- > Linea\ (line+m)\ rec)\ d)$
$\overset{ppio.\ de\ reemplazo}{=} (line\ rec- > Linea\ (line+m))\ k\ (indentar\ m\ d)$
$\overset{\beta}{=} (rec- > Linea\ (k+m)\ rec)\ (indentar\ m\ d)$
$\overset{\beta}{=} Linea\ (k+m)\ (indentar\ m\ d)$
$\overset{conmutatividad\ de\ la\ suma}{=} Linea\ (m+k)\ (indentar\ m\ d)$
Como se quería probar.

4.1.4 Demostración Ejercicio

$$\texttt{indentar}\ n(\texttt{indentar}\ m\ x) = \texttt{indentar}\ (n+m)\ x$$

Hacemos inducción estructural sobre el tipo Doc. Lo recordamos :

```
data Doc = Vacio | Texto String Doc | Linea Int Doc
```

$\forall n, m :: Z^+.\ x :: Doc$
Definimos :

1. **Caso Base :**

$$P(Vacio)$$
$$indentar\ n\ (indentar\ Vacio) = indentar\ (n+m)\ Vacio$$
$$\overset{I_1}{\equiv} indentar\ n\ (Vacio) = indentar\ (n+m)\ Vacio$$
$$\overset{Lema_1}{\equiv} Vacio = Vacio$$

2. **Caso Inductivo :** Vamos a presentar entonces la **Hipótesis Inductiva.**

$$\forall d :: Doc.\ indentar\ n\ (indentar\ m\ d) = indentar\ (n+m)\ d$$

Para el caso de este tipo de datos tenemos dos casos :

(a) $\forall d :: Doc. \forall s :: String.\ P(d) \implies P(Texto\ s\ d)$
(b) $\forall d :: Doc. \forall i :: Z^+.\ P(d) \implies P(Linea\ i\ d)$

Vamos a comenzar.

(a)

$$indentar\ n\ (indentar\ m\ (Texto\ s\ d)) = indentar\ (n+m)\ (Texto\ s\ d)$$
$$\overset{Lema_2}{\equiv} indentar\ n\ (Texto\ s\ (indentar\ m\ d)) = indentar\ (n+m)\ (Texto\ s\ d)$$
$$\overset{Lema_2}{\equiv} Texto\ s\ (indentar\ n\ (indentar\ m\ d)) = indentar\ (n+m)\ (Texto\ s\ d)$$
$$\overset{HI}{\equiv} Texto\ s\ (indentar\ (n+m)\ d)) = indentar\ (n+m)\ (Texto\ s\ d)$$
$$\overset{HI}{\equiv} indentar\ (n+m)\ (Texto\ s\ d) = indentar\ (n+m)\ (Texto\ s\ d)$$

Como se quería probar.

(b)

$$indentar\ n\ (indentar\ m\ (Linea\ i\ d)) = indentar\ (n+m)\ (Linea\ i\ d)$$
$$\overset{Lema_3}{\equiv} indentar\ n\ (Linea\ (m+i)\ (indentar\ m\ d)) = indentar\ (n+m)\ (Linea\ i\ d)$$
$$\overset{Lema_3}{\equiv} Linea\ (n+m+i)\ (indentar\ n\ (indentar\ m\ d)) = indentar\ (n+m)\ (Linea\ i\ d)$$
$$\overset{HI}{\equiv} Linea\ (n+m+i)\ (indentar\ (n+m)\ d) = indentar\ (n+m)\ (Linea\ i\ d)$$
$$\overset{Lema_3}{\equiv} indentar\ (n+m)\ (Linea\ i\ d) = indentar\ (n+m)\ (Linea\ i\ d)$$

Como se quería probar.