Teórica 0 : Introducción a la Programación Funcional

Tomás Felipe Melli

March 22, 2025

Índice

1 Aspectos de los lenguajes de Programación

1. **Programación** : Los lenguajes tienen diferentes características como:

   - **typing** (cómo se manejan los tipos de datos) que puede ser *dynamic* (la variable se determina en runtime (si escribimos en python algo como : a = 0) y el tipo puede cambiar durante la ejecución esto incurre en problemas de tipo en runtime) o *static*(que se define en tiempo de compilación (si escribimos en c++ algo como : int a = 0;) y se debe declarar explícitamente. Los errores de tipo se detectan antes de la ejecución.).
   - **Administración de la memoria** : hay lenguajes en los que el programador es responsable de asignar y liberar, como **C**(int *ptr = (int*) malloc(sizeof(int));), denominados lenguajes de **administración manual** y otros en los cuales la **administración es automática**, como JAVA, Python,... en los cuales se implementa un **Garbage Collector**, un programa responsable de liberar espacio ocupado por objetos que ya no se usan, libera memoria y evita accesos a *dangling pointers*(punteros a objetos ya liberados). Este programa es un componente del **runtime environment(RTE)** del lenguaje de programación. Esto es, el conjunto de herramientas y recursos que un programa necesita para ejecutarse correctamente. Esto inlcuye *cargar el código (traducción a un lenguaje que entienda la máquina)*, *administrar la memoria* (con o sin GC), *interaccción con el sistema*(los archivos con los que interactúa el programa), ...
   - **Forma de almacenar funciones** : los lenguajes más antiguos almacenaban las funciones en la **sección de código(text segment)** con la intención de programar de forma estructurada y donde las funciones son bloques de código estáticos, no valores manipulables. Además, hay factores de rendimiento y compatibilidad. Esto nos lleva a que lenguajes como C,Pascal, Fortran consideren a las funciones como de **primer orden**. Por el contrario, lenguajes como Python, Haskell, entre otros, las funciones se almacenan en el **heap**, es decir, considera a *las funciones como objetos manipulables* y sus nombres se referencian a esas direcciones de memoria que pueden ser modificadas en runtime, ser argumentos de otras funciones o ser devueltas como resultado de una función. Estos lenguajes tratan a sus funciones como **funciones de orden superior (First-Class Functions)**.
   - **Mutabilidad** : se refiere a la capacidad que tiene un objeto de cambiar luego de ser creado. Esta capacidad está estrechamente vinculada a la forma en la que los lenguajes gestionan los objetos y la memoria. Los lenguajes que permiten **mutabilidad** pueden modificar estructuras de datos **sin necesidad de crear instancias nuevas**.(Ejemplos : C/C++, Python). Por otro lado, existen lenguajes en los que **un objeto es Inmutable**, es decir, su estado no puede cambiar luego de su creación. Para modificarlo es necesario crear una nueva instancia. (Ejemplos : Haskell, strings en Java, tuplas y cadenas en Pyhton, Scala). Se utiliza esta forma de tratar a los objetos para promover la seguridad, facilitar la programación concurrente, entre otras cosas.
   - **Scope (Modelo de Alcance)** en un lenguaje de programación define **cómo las variables son accedidas y resueltas**. Existen lenguajes de **alcance estático** en los que las variables se resuelven en tiempo de compilación según cómo se estructure el código (el compilador sabe en qué bloques se pueden acceder). Lo bueno es que se entiende qué variables estpan disponible en cierto bloque. Lo malo es que si necesitamos modificar el scope en cierta situación dinámicamente, no se puede. (Ejemplos : C/C++, Python). Por otra parte, tenemos lenguajes en los que el **alcance es dinámico** y la resolución de las variables ocurre en runtime según el contexto, es decir, si una variable ya fue definida por otra función, otra que se ejecute luego, podrá usarla. La complicación de esto es que debuggear es un bardo y entender el programa, más.
   - **Resolución de Nombres** : es el proceso por el cual un lenguaje de programación determina a qué variables o funciones se hace referencia en el código. Tenemos el **Early Binding**(o Resilución Temprana) que el enlace entre una variable o función y su valor o dirección se realiza en tiempo de compilación o en etapas anteriores a la ejecución del programa. (Ejemplos : C, Java). Esto hace que el programa se ejecute más rápido y se comporte de forma más predecible. Por el contrario, el **Late Binding**(o Resolución Tardía) ocurre en runtime. Esto hace que el programa no conozca las referencias exactas a funciones o variables hasta que se ejecuta el código. Esto hace que las direcciones de las funciones puedan cambiar en función del contexto de ejecución. La motivación es permitir mayor flexibilidad para modificaciones dinámicas incurriendo en la carga adicional en runtime.
   - **Inferencia de tipos** : es el proceso por el cual un compilador o intérprete determina el tipo de una variable o expresión de manera automática, sin necesidad de que el programador lo especifique explícitamente en el código. Para ello, el compilador o el intérprete puede deducir el tipo de una variable basándose en el valor asignado, las operaciones realizadas o el contexto en el que se usa. Esto lo puede lograr gracias al **sistema de tipos** que es un módulo del mismo formado por el componente de inferencia de tipos, verificación de consistencia de tipos, resolución de nombres y gestión de las conversiones de tipos. En este item en particular nos interesa ver que hay lenguajes que tienen **inferencia de tipos estática** en donde la inferencia ocurre en *tiempo de compilación* como lo hace Haskell y Scala. Como contrapartida, existen lenguajes que los hacen de maner **dinámica** en runtime como Python. Lo bueno de inferirlo es que hace el código más limpio y ayuda a reducir los errores de tipo porque el compilador lo verifica sin necesidad de que el programador lo haga. Lo malo es que el código puede no ser claro

en el sentido que si queremos depurarlo los tipos no son claros. Puede generar ambiguedad en algunos casos si el compilador no infiere correctamente.

- El **Determinismo** es un concepto que se vincula al comportamiento de un programa en ejecución, en particular, a *la forma en la que un lenguaje maneja el flujo de control y la gestión de estado*. Los lenguajes **Deterministas** como Haskell son aquellos que dada una entrada específica, produce siempre el mismo resultado en cada ejecución, sin importar cuántas veces se ejecute. Por el contrario, lenguajes **No Deterministas** que permiten *ejecución asincrónica* o *entornos concurrentes* el comportamiento puede variar en ejecuciones succesivas cuando la entrada no cambia. Esta desición dependerá enteramente en el modelo de ejecución que cada lenguaje promueve y de las características que el lenguaje permite (como concurrencia, asincronía o aleatoriedad).
- **Pasaje de parámetros** : es la manera en la que las funciones manejan los argumentos que se les pasan. El **pasaje por copia** ocurre cuando el parámetro es una copia del valor de la variable y su manipulación no afecta el valor original de la variable (la función opera sobre una copia local del dato). Esto puede ser un problema si se pasa una estructura enorme ya que es costoso en memoria y tiempo. Por otro lado, existe el **pasaje por referencia** que los que se le pasa a la función es una **referencia (dirección de memoria** y por ello, cualquier modificación, impacta directo en memoria.
  Estas maneras están vinculadas a cómo el lenguaje maneja tipos de datos (especialmente tipos primitivos y objetos mutables), y a sus características semánticas. No se puede elegir libremente en la mayoría de los lenguajes ya que está definido **por tipo y reglas del lenguaje**. No está demás mencionar en C++ el uso de *& o *¨*. Dicho esto, el modelo de pasaje de parámetros es predefinido por el diseño del lenguaje y está determinado por las características del lenguaje, como los tipos de datos, la gestión de memoria y las optimizaciones que se hacen a nivel de compilador.
- **Evaluación de expresiones** : existen lenguajes de **evaluación estricta (Strict)** como C

```c
int suma(int x, int y) {
    return x + y;
}
int main() {
    int a = 5, b = 10;
    suma(a + 1, b * 2);
    // ‘a + 1‘ se eval a como 6 y ‘b * 2‘ se eval a como 20 antes de llamar a la
        funci n.
    return 0;
}
```

que evalúa inmediatamente (momento de ejecución), en este caso, cuando se hace el llamado a la función. En caso contrario, lenguajes como Haskell de **evaluación diferida (Lazy)**

```haskell
suma x y = x + y

main = print (suma (1 + 1) (2 * 2))
-- ‘1 + 1‘ y ‘2 * 2‘ se eval an solo cuando la funci n ‘suma‘ las necesita.
```

no se evalúa inmediatamente, en este caso, cuando la función necesita el valor. Esta forma de evaluar expresiones (Lazy) postpone la evaluación de las expresiones hasta que sean realmente utilizadas.

- **Sistema de Tipos** : determinan características avanzadas sobre cómo construir y manipular los tipos. Ya hablamos de la mutabilidad y la evaluación, ahora toca ver **el nivel de expresividad** que está vinculado a la capacidad de abstracción que tiene cierto lenguaje. Este alto grado de abstracción permite tipos complejos como :
  - **Tipos de Datos Inductivos** : los tipos inductivos son tipos de datos que se definen de manera recursiva. Un tipo inductivo es un tipo de datos que puede ser construido a partir de un conjunto de casos base y un conjunto de reglas de construcción. En Haskell por ejemplo, un tipo de dato inductivo para una lista puede ser definido como:

    ```haskell
    data List a = Empty | Cons a (List a)
    ```

  - **Tipos Co-Inductivos** : Son tipos de datos definidos de manera recursiva pero infinita, es decir, sus valores pueden ser generados de forma perezosa (lazy), sin necesidad de ser completamente evaluados en el momento de la definición. Un tipo de dato co-inductivo podría ser un stream (una lista infinita), que se define de forma recursiva:

    ```haskell
    data Stream a = Cons a (Stream a)
    ```

  - **GADTs (Generalized Algebraic Data Types)** : Permiten que los constructores de datos tengan tipos más específicos y detallados, en lugar de ser generales. En Haskell, un GADT para representar expresiones matemáticas podría ser:

```haskell
data Expr a where
    Val  :: Int -> Expr Int
    Add  :: Expr Int -> Expr Int -> Expr Int
    BoolVal :: Bool -> Expr Bool
    And :: Expr Bool -> Expr Bool -> Expr Bool
```

- **Familias de Tipos Dependientes** : Las familias dependientes de tipos son una extensión avanzada de los tipos en los que los tipos dependen de valores. En lugar de tener un tipo genérico como List a, puedes tener un tipo que dependa de un valor, como una lista de longitud n. Este es un concepto central en lenguajes con dependencias de tipos

- En algunos lenguajes de programación existen dos conceptos fundamentales:

  (a) **Pattern Matching** : es una técnica que permite desestructurar y comparar estructuras de datos con patrones definidos en el código para extraer valores o realizar acciones basadas en la estructura de esos datos. Se define un patrón que se compara contra una variable, si coincide, se ejecuta el código correspondiente. El pattern matching puede ser secuencial, evaluando los patrones de arriba hacia abajo hasta encontrar una coincidencia. Veamos un ejemplo en Haskell :

```haskell
data Shape = Circle Float | Rectangle Float Float

area :: Shape -> Float
area (Circle r) = pi * r * r
area (Rectangle l w) = l * w
```

  (b) **Unificación** : es un proceso utilizado para resolver variables dentro de expresiones o patrones. Consiste en encontrar una correspondencia entre dos expresiones de manera que las variables involucradas obtengan un valor común.

```haskell
-- Calcular el  rea  de un c rculo con radio 5.0
area (Circle 5.0)
```

  Lo que sucederá es que Haskell intenta unificar el valor Circle 5.0 con el patrón (Circle r)

  Habiendo dicho esto, internamente, Haskell unidica el valor Circle 5.0 con el patrón (Circle r), se establece r = 5.0, coincide con el patrón (Circle r) y se calcula pi * r * r, lo que da el área del círculo. Pattern matching es el proceso en el que Haskell compara la entrada (en este caso, una Shape) con los patrones (Circle r) o (Rectangle l w). Unificación ocurre cuando se comparan esos valores con las variables en los patrones, como unificar Circle 5.0 con (Circle r) para obtener r = 5.0.

- El **polimorfismo paramétrico** es un concepto importante en lenguajes de programación funcional (y otros lenguajes también) que se refiere a la capacidad de escribir funciones y tipos de datos que pueden trabajar con cualquier tipo de dato, sin necesidad de saber de antemano el tipo específico. Se utiliza principalmente con tipos genéricos (parametrizados) y es una de las características clave de Haskell. Permite que las funciones o tipos sean independientes del tipo de dato con el que operan. Ejemplo de la función identidad en Haskell.

```haskell
id :: a -> a
id x = x
```

  La notación **a -¿ a** significa que la función puede operar sobre **cualquier tipo**.
  Esta característica del lenguaje permite **independencia de tipos, seguridad de tipo, reutilización de código.**

- **Subclasificación y Polimorfismo de Subtipos**:
  - La subclasificación es un concepto en programación orientada a objetos (OOP) que define una relación entre clases o tipos, donde una clase (subclase) hereda las propiedades y comportamientos (métodos) de otra clase (superclase). En otras palabras, una subclase es un tipo especializado de su superclase, lo que le permite reutilizar el código de la superclase y, en algunos casos, sobrescribir (o extender) sus métodos.
  - El polimorfismo de subtipos es un tipo de polimorfismo que ocurre cuando una variable de un tipo más general (superclase) puede hacer referencia a un objeto de un tipo más específico (subclase). Esto permite que las funciones o métodos que esperan un tipo general puedan operar con objetos de cualquier tipo que sea un subtipo de ese tipo general.

- **Herencia** : La herencia es un mecanismo mediante el cual una clase puede heredar características (propiedades y métodos) de otra clase. Hay dos tipos principales de herencia:

  (a) En la herencia simple, una clase hereda de una sola clase base o superclase.

  (b) En la herencia múltiple, una clase puede heredar de varias clases base o superclases. Esto permite una mayor reutilización del código, ya que una subclase puede combinar comportamientos de diferentes clases. Sin embargo, también puede hacer que el diseño sea más complicado y dar lugar a conflictos de ambigüedad (por ejemplo, si dos superclases definen el mismo método).

- **Estructuras de Control No Local** : son mecanismos que permiten alterar el flujo de ejecución de un programa de manera no secuencial o no local. Es decir, permiten saltar fuera del contexto de ejecución actual (por ejemplo, salir de varias capas de llamadas a funciones) y transferir el control a una parte diferente del programa, sin seguir la secuencia de ejecución normal. Esto está estrechamente vinculado con el **paradigma de programación y lenguaje**.
  - En los lenguajes **imperativos** (como C, Java, Python, C++), el control de flujo suele ser explícito y secuencial. Sin embargo, los lenguajes imperativos también permiten mecanismos de control no local como excepciones, goto, y salidas anticipadas (return, break).
  - Los lenguajes **funcionales** (como Haskell, Lisp, Scala) adoptan un enfoque muy diferente hacia el control de flujo. En lugar de realizar operaciones sobre el estado del programa (como en los lenguajes imperativos), el control no local en lenguajes funcionales está más relacionado con la evaluación lazy (ya que el flujo de ejecución depende de cuándo y cómo se necesitan los resultados), las **mónadas** y el manejo de efectos secundarios. Una mónada es un patrón de diseño que proporciona una manera de manejar cálculos secuenciales en un contexto que puede implicar efectos secundarios preservando la pureza funcional. Ejemplo de **Maybe**, una mónada que se usa para representar cálculos que pueden fallar. Puede ser Nothing (cuando el cálculo falla) o Just a (cuando el cálculo tiene éxito).:

```haskell
data Maybe a = Nothing | Just a
```

  - En lenguajes **orientados a objetos** (como Java, C++, Python), el control de flujo no local es importante, especialmente en el manejo de excepciones. Los lenguajes orientados a objetos permiten que el flujo de ejecución se altere mediante excepciones y manejadores de eventos. El uso de **try/catch** para capturar excepciones, lo que ofrece control no local sobre cómo y dónde se manejan los errores, sin necesidad de interrumpir el flujo de ejecución de manera manual.
  - En lenguajes **lógicos** como Prolog, el control no local se maneja de una forma muy diferente. El flujo de ejecución está basado en la resolución de hechos y reglas. La unificación y la búsqueda de soluciones en estos lenguajes pueden considerarse una forma de control no local, ya que no siguen el flujo secuencial de un programa imperativo. El motor de inferencia es responsable de decidir el orden en el que se exploran las posibles soluciones. El **backtracking** es una técnica fundamental en la programación lógica que permite al sistema "deshacer" una decisión y probar otra opción, lo que permite el control no local.

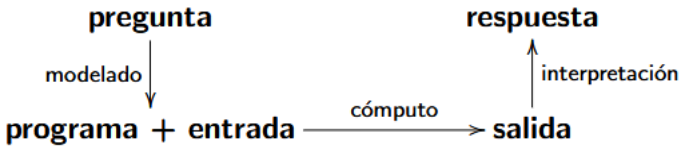
2. **Semántica** : cómo se comportan los programas escritos en ese lenguaje ? cómo las construcciones del lenguaje (como instrucciones, expresiones, estructuras de control, funciones, etc.) deben comportarse cuando un programa se ejecuta. Existen diferentes tipos :

   - **Semántica Operacional** : Define las reglas de evaluación para las expresiones y cómo las instrucciones afectan al estado de la máquina (memoria, registros, etc.). Es como una "máquina abstracta" que toma un programa y lo ejecuta paso a paso.
   - **Semántica Axiomatica** : se basa en reglas formales y lógicas para definir cómo deben comportarse los programas. En lugar de describir la ejecución paso a paso, describe lo que debe ser cierto antes y después de la ejecución de un fragmento de código. Esto se utiliza principalmente en el contexto de la verificación formal y correctitud de programas, donde queremos garantizar que un programa haga lo que se espera de él, independientemente de los detalles de ejecución.
   - **Semántica Denotacional** : se enfoca en la interpretación matemática de los programas. Asocia cada constructo del lenguaje a una función matemática que describe su comportamiento. Es más abstracta que la semántica operacional y no describe cómo se ejecuta el programa, sino cómo se evalúa o denota su significado.
   - **Semántica de la Tipificación** : En lenguajes con sistemas de tipos, como Haskell o Java, la semántica de la tipificación define cómo los tipos de datos afectan el comportamiento de un programa. La semántica de la tipificación especifica qué significa que una variable o función tenga un tipo determinado y cómo esos tipos se pueden combinar o transformar en el contexto de la ejecución del programa.

3. **Implementación** : se refiere al ¿Cómo es capaz de ejecutar programas escritos en otros lenguajes?.

2 Programación Funcional

Un problema central de la computación es el de **procesar la información** :



La **programación funcional** es un **paradigma** de programación que se basa en la definición y aplicación de funciones para resolver problemas. Las funciones no son simplemente bloques de código que ejecutan acciones. Las **Funciones verdaderas (parciales)** deben cumplir ciertas propiedades matemáticas :

- **No tiene efectos secundarios** : Al aplicar una función, no se modifica ningún estado o variable fuera de la propia función. Esto significa que las funciones son predecibles, y su salida depende únicamente de sus entradas.
- **Determinismo** : Para una misma entrada, siempre se obtiene la misma salida.
- **Inmutabilidad** : Las estructuras de datos no se pueden modificar.

En este paradigma, las funciones son tratadas como datos. Esto significa que las funciones pueden:

- Pasarse como parámetros
- Devolver funciones como resultados (funciones de orden superior como ya vimos)
- Formar parte de estructuras de datos: Por ejemplo, en un árbol binario, los nodos pueden contener funciones, y no solo valores.

2.1 Programa Funcional

Un programa funcional está dado por un **conjunto de ecuaciones orientadas**:

```
1 longitud [] = 0
2 longitud (x:xs) = 1 + longitud xs
```

Entonces, si queremos evaluar **longitud [10,20,30]** :

$$= \text{longitud}(10 : (20 : (30 : [])))$$

$$= 1 + \text{longitud}(20 : (30 : []))$$

$$= 1 + (1 + \text{longitud}(30 : []))$$

$$= 1 + (1 + (1 + \text{longitud}[]))$$

$$= 1 + (1 + (1 + 0))$$

$$= 1 + (1 + 1)$$

$$= 1 + 2$$

$$= 3$$

2.2 Expresiones

Las expresiones son secuencias de símbolos que sirven para representar datos, funciones y funciones aplicadas a los datos. (Las funciones también son datos). Pueden ser de distintos tipos :

- **Constructores** : definen los tipos de datos y cómo podemos crear valores de esos tipos.

```
True  -- Constructor del tipo de datos Booleano
[]    -- Constructor de lista vac a
5     -- Constructor de tipo num rico
1 : [2, 3]  -- Usando el constructor ‘(:) llamado cons‘ para crear una lista: [1, 2, 3]
```

- **Variables** : nombres que representan valores en una expresión.
  - Variables simples: x, xs, longitud, ordenar
  - Funciones: +, *, not
- **Aplicación de Expresiones** : aplicar una función a uno o más valores. En Haskell, la aplicación de funciones es implícita.
  - Aplicación de funciones simples:

```
ordenar lista      -- Aplica la funci n 'ordenar' a la lista 'lista'
not True           -- Aplica la funci n 'not' a 'True', resultado: False
not (not True)     -- Aplica 'not' a 'True' dos veces, resultado: True
(+) 1 2            -- Suma 1 y 2 usando la funci n '+', resultado: 3
```

  - Aplicación de funciones con otros resultados:

```
((+) 1) (alCuadrado 5)  -- Aplica la funci n '(+)' con 1 al resultado de 'alCuadrado 5'
```

- Otras Formas de Expresiones:
  - Expresiones condicionales (if):

```
if x > 18 then "Mayor" else "Menor"
```

  - Pattern Matching

2.2.1 Aplicación de Funciones Asociativa a Izquierda

En Haskell, la aplicación de funciones es asociativa a la izquierda. Esto significa que **cuando una función se aplica a múltiples argumentos sin paréntesis, la evaluación se agrupa de izquierda a derecha.**

$$f\ a\ b\ c\ d \equiv (((f\ a)\ b)\ c)\ d$$

Sin embargo, **no todas las operaciones son izquierda-asociativas**, por ejemplo **(:) cons** en listas es asociativo a derecha :

```
(:) :: a -> [a] -> [a]
```

Las listas están definidas

```
data [a] = [] | a : [a]
```

Ejemplo :

$$1 : 2 : 3 : []$$

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

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

$$\equiv [1, 2, 3]$$

2.3 Tipos

Un **tipo** es una especificación del invariante de un dato o de una función.

```
99 :: Int
not :: Bool -> Bool
not True :: Bool
(+) :: Int -> (Int -> Int)
(+) 1 :: Int -> Int
((+) 1) 2 :: Int
```

2.3.1 Condiciones de tipado

En Haskell, las condiciones de tipado son reglas que aseguran que un programa sea consistente en términos de los tipos de sus expresiones y operaciones. Para que un programa esté bien tipado en Haskell, debe cumplir con las siguientes condiciones:

1. Todas las expresiones deben tener tipo.
2. Cada variable se debe usar siempre con un mismo tipo.
3. Los dos lados de una ecuación deben tener el mismo tipo.
4. El argumento de una función debe tener el tipo del dominio.
5. El resultado de una función debe tener el tipo del codominio.

En programas bien tipado :

$$\frac{\texttt{f :: a -> b} \qquad \texttt{x :: a}}{\texttt{f x :: b}}$$

Si la función **f** recibe un valor de tipo **a** y evaluada devuelve uno de tipo **b**, si **x es de tipo a** $\Longrightarrow$ **f x** evaluada devuelve un valor de tipo b.

2.4 Asociación de Tipos Asociativa a derecha

Convenimos que $\rightarrow$ es asociativo a derecha

$$a -> b -> c -> d \equiv a -> (b -> (c -> d))$$

Esto se puede entender como que **f** toma un argumento de tipo **a**, y **devuelve una función que toma un argumento de tipo b** y devuelve otra función que toma un argumento de tipo c y finalmente devuelve un valor de tipo d.

2.5 Polimorfismo

Ya hablamos de polimorfismo en Haskell. Miremos **flip**. La función **flip** invierte el orden de los argumentos de una función binaria. Si f es una función que toma dos argumentos, flip devolverá una nueva función que toma los mismos dos argumentos pero en orden inverso.

```
flip :: (a -> b -> c) -> (b -> a -> c)
```

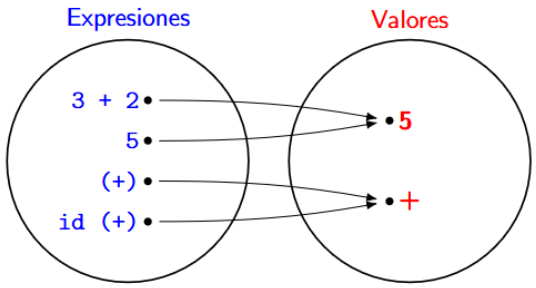
Flip toma una función de tipo (a $\rightarrow$ b $\rightarrow$ c) y la convierte en una función de tipo (b $\rightarrow$ a $\rightarrow$ c)
Ejemplo :

```
flip f x y = f y x
```

$$\begin{aligned} &\text{flip}\,(:)\,[2,3]\,1 \\ &\equiv (:)\,1\,[2,3] \\ &\equiv 1 : [2,3] \\ &= [1,2,3] \end{aligned}$$

2.6 Modelo de cómputo

El modelo de cómputo de Haskell se basa en el uso de expresiones (combinación de valores y operadores) y valores (evaluación de una expresión) para representar el proceso de computación.



Existen expresiones bien tipadas que *no tienen valor* como la división por cero, lo que resulta en una operación indefinida. En Haskell, este valor se representa como $\perp$ (bottom), que indica que la expresión no tiene un valor definido.
Ya mencionamos que un programa funcional está dado por un conjunto de ecuaciones **orientadas** que definen las relaciones entre diferentes expresiones. Este enfoque se puede abordar desde dos puntos de vista diferentes: el **denotacional** y el **operacional**.

1. Punto de vista **denotacional** : En este enfoque, una ecuación como $e1 = e2$ se interpreta como una declaración de equivalencia. Es decir, se afirma que las expresiones $e1$ y $e2$ tienen el mismo significado o la misma denotación (valor). No se está enfocando en cómo se evalúan las expresiones, sino en que ambas representan el mismo resultado o propiedad.

2. Punto de vista **operacional** : Desde este enfoque, una ecuación como $e1 = e2$ se interpreta como una reducción o proceso de evaluación. Esto significa que para computar el valor de $e1$, se puede reducir o sustituir $e1$ por $e2$ y seguir con el proceso de computación de $e2$.
   Acá no estamos hablando de qué significan las expresiones, sino de cómo se calculan sus valores. Por ejemplo, si $e1$ es una expresión compleja, el proceso de computación se puede descomponer en pasos sucesivos donde cada paso utiliza las ecuaciones definidas en el programa.

Ejemplo :

```
-- Definimos una funci n ‘f‘ en t rminos de una ecuaci n
f x = x + 1

-- Desde el punto de vista denotacional:
-- f 2 tiene el mismo valor que 2 + 1

-- Desde el punto de vista operacional:
-- Para computar ‘f 2‘, reemplazamos ‘f x‘ por ‘x + 1‘, obteniendo:
f 2 = 2 + 1 = 3
```

En los lenguajes de programación funcional, las ecuaciones deben seguir ciertas restricciones para ser consideradas sintácticamente válidas, especialmente cuando se usan **patrones** en el lado izquierdo de la ecuación. El lado izquierdo de la ecuación debe estar formado por una función aplicada a patrones específicos. Un patrón puede ser:

- Una **variable**
- Un **comodín (_)** : Se usa para denotar que no nos interesa el valor de la parte que coincide con el comodín.
- Un **constructor aplicado a patrones** : Esto implica que un constructor (como :, [], o Just) se aplica a patrones, es decir, se descompone la estructura de datos para hacer coincidir sus componentes.

Además, hay una regla importante para asegurar que el lado izquierdo de una ecuación esté correctamente formado: no debe haber variables repetidas en el mismo patrón. Esto es porque las variables en los patrones deben corresponder a valores diferentes, y si una variable se repite, se estaría tratando de asociar el mismo valor a diferentes partes de una estructura, lo cual no es válido.
Ejemplos :

```
sumaPrimeros (x : y : z : _) = x + y + z
predecesor (n + 1) = n
iguales x x = True
```

1. El patrón x : y : z : _ es una lista con al menos tres elementos, donde x, y, y z son variables que coinciden con los tres primeros elementos de la lista, y el _ es un comodín que ignora el resto de la lista. Es válido porque no hay variables repetidas y el patrón está descomponiendo la lista correctamente.

2. Este patrón no está bien formado en Haskell. Aunque n + 1 parece un patrón que intenta descomponer una expresión matemática, no es una forma válida de patrón en Haskell, porque no se puede aplicar un operador directamente en un patrón. Los patrones deben ser funciones aplicadas a valores, no operaciones aritméticas. No es válido porque n + 1 no es un patrón adecuado.

3. La variable x se repite en el mismo patrón, lo cual no es permitido. Haskell no permite que una variable aparezca dos veces en un patrón porque implicaría que las dos instancias de x deben tener el mismo valor, lo cual no tiene sentido en este contexto. No es válido porque x está repetida en el patrón.

Pasos para evaluar una expresión :

1. Buscar la subexpresión que coincide con el lado izquierdo de una ecuación: se identifican las funciones o constructores que están aplicados a valores y se buscan las reglas de la función que puedan ser aplicadas a esa forma.

2. Reemplazar la subexpresión por el lado derecho de la ecuación: Una vez que se encuentra la subexpresión correspondiente a un lado izquierdo de una ecuación, se reemplaza esa subexpresión por el valor que aparece en el lado derecho de la ecuación. El lado derecho suele ser una expresión que describe el comportamiento o resultado de la operación.

3. Continuar evaluando la expresión resultante: Después de hacer el reemplazo, la evaluación no termina; la nueva expresión resultante debe ser evaluada de acuerdo con el mismo proceso. Si la expresión sigue siendo compleja (es decir, no está completamente evaluada), se repiten los pasos anteriores hasta que la evaluación llegue a un valor.

Cuándo se detiene la evaluación?

- **La expresión es un constructor o un constructor aplicado**: Si la expresión no puede ser reducida más (es decir, es una expresión completa o un valor), la evaluación se detiene. Los valores como True, False, listas, tuplas, y constructores aplicados son ejemplos de esto.

- **La expresión es una función parcialmente aplicada**: Si la expresión es una función parcialmente aplicada (es decir, una función que no tiene todos los argumentos necesarios para ser completamente evaluada), la evaluación también se detiene, ya que no se puede hacer más sin aplicar más argumentos.

- Se alcanza un estado de error: Si la expresión no coincide con ninguna ecuación definida, o si se encuentra con una expresión no válida (por ejemplo, la división por cero), la evaluación se detendrá en un estado de error.

Ejemplos de lo anterior:

- **resultado — constructor**

```haskell
tail :: [a] -> [a]
tail (_ : xs) = xs
```

$$\text{tail}(\text{tail}[1,2,3]) \quad \Rightarrow \quad \text{tail}[2,3] \quad \Rightarrow \quad [3]$$

- **resultado — función parcialmente aplicada**

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

$$\text{const } (\text{const } 1)\ 2 \quad \Rightarrow \quad \text{const } 1$$

- **indefinición — error**

```haskell
head :: [a] -> a
head (x : _) = x
```

$$\text{head}(\text{head}[[],[1],[1,1]])$$

1. Primero, evaluamos head$[[],[1],[1,1]]$:

$$\text{head}[[],[1],[1,1]] \quad \Rightarrow \quad []$$

2. Luego, evaluamos head$[]$, lo que nos da un error ya que head no está definida para listas vacías.

$$\text{head}[] \quad \Rightarrow \quad \bot$$

- **indefinición - no terminación**

```
loop :: Int -> a
loop n = loop (n + 1)
```

$$\begin{aligned} \text{loop0} &\Rightarrow \text{loop}(1+0) \\ &\Rightarrow \text{loop}(1+(1+0)) \\ &\Rightarrow \text{loop}(1+(1+(1+0))) \\ &\Rightarrow \dots \end{aligned}$$

Este proceso sigue indefinidamente, ya que la función `loop` nunca alcanza un valor base y sigue aplicándose recursivamente.

- **evaluación no estricta**
Definimos una expresión que se refiere a sí misma:

```
indefinido :: Int
indefinido = indefinido
```

$$\text{head}(\text{tail}[\text{indefinido}, 1, \text{indefinido}])$$

Primero, evaluamos la función `tail`:

$$\text{tail}[\text{indefinido}, 1, \text{indefinido}] \Rightarrow [1, \text{indefinido}]$$

Luego, evaluamos la función `head`:

$$\text{head}[1, \text{indefinido}] \Rightarrow 1$$

Por lo tanto, la evaluación de la expresión resulta en:

$$\text{head}(\text{tail}[\text{indefinido}, 1, \text{indefinido}]) = 1$$

- **listas infinitas**
Definimos la función `desde` que genera una lista infinita:

```
desde :: Int -> [Int]
desde n = n : desde (n + 1)
```

$$\begin{aligned} &\text{desde } 0 \\ \Rightarrow\ &0 : \text{desde } 1 \\ \Rightarrow\ &0 : (1 : \text{desde } 2) \\ \Rightarrow\ &0 : (1 : (2 : \text{desde } 3)) \Rightarrow \dots \end{aligned}$$

Ahora, evaluamos una expresión que utiliza la función `head` y `tail` sobre la lista infinita:

$$\text{head}(\text{tail } (\text{desde } 0))$$

Evaluamos paso a paso:

$$\begin{aligned} &\text{head } (\text{tail } (0 : \text{desde } 1)) \\ \Rightarrow\ &\text{head } (\text{desde } 1) \\ \Rightarrow\ &\text{head}(1 : \text{desde } 2) \\ \Rightarrow\ &1 \end{aligned}$$

El **orden de las ecuaciones en Haskell es importante.** Las ecuaciones se prueban en orden, y cuando se encuentra una coincidencia, Haskell utiliza esa ecuación y no evalúa las siguientes, incluso si podrían coincidir también. Ejemplo :

```haskell
esCorta :: [a] -> Bool
esCorta (_ : _ : _) = False
esCorta _ = True
```

1. **Primera ecuación** : define que una lista con al menos dos elementos (_ : _ : _) devuelve False. El patrón _ : _ : _ coincide con cualquier lista que tenga dos o más elementos. Es un comodín que representa una lista con al menos dos elementos, por lo que la función devuelve False para estas listas.

2. **Segunda ecuación** : Esta ecuación cubre todos los demás casos, devolviendo True. Esta ecuación se aplica cuando la lista no tiene dos o más elementos.

$$\begin{aligned} \text{esCorta}[] &\Rightarrow \text{True} \\ \text{esCorta}[1] &\Rightarrow \text{True} \\ \text{esCorta}[1,2] &\Rightarrow \text{False} \end{aligned}$$

2.7 Funciones de orden superior

Ya hablamos un poco de esto.Son aquellas que toman otras como argumentos o devuelven funciones como resultados. Vamos a definir la **composición de funciones (”g . f”)**

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

Usando **notación lambda** :

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

Qué tienen en común las siguientes funciones ?

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

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

longitudL :: [[a]] -> [Int]
longitudL [] = []
longitudL (x : xs) = length x : longitudL xs
```

Todas siguen este esquema :

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

1. – **caso base** : lista vacía

2. – **caso inductivo** : cuando la lista tiene al menos un elemento (x : xs), se aplica la función f al primer elemento (x), y luego se recurre al resto de la lista (xs) utilizando la misma función g.

Qué relación hay entre las siguientes funciones ?

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

noVacias :: [[a]] -> [[a]]
noVacias [] = []
noVacias (x : xs) = if not (null x)
                        then x : noVacias xs
                        else noVacias xs
```

Todas siguen este esquema :

```haskell
g [] = []
g (x : xs) = if p x
                then x : g xs
                else g xs
```

1. – **caso base** : lista vacía
2. – **caso inductivo** : cuando la lista tiene al menos un elemento (x : xs), se evalúa la condición p x. Si la condición es verdadera, entonces se agrega el elemento x al resultado y se continúa procesando el resto de la lista (xs). Si la condición es falsa, el elemento x se omite y se continúa con el resto de la lista.

Cómo se puede abstraer este esquema ?
La función **filter** en Haskell realiza exactamente este proceso de filtrado, y se define :

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

Donde **p** es un **predicado de tipo (a → Bool)** que es una función que recibe un elemento de tipo a y devuelve un valor de tipo Bool (es decir, True o False). $[a]$ es la lista que se filtra. Y la función devuelve una lista del tipo $[a]$ que contiene solo aquellos elementos de la lista original que cumplen con el **predicado p**.
Con el uso de filter, se puede aplicar a :

```haskell
negativos :: [Int] -> [Int]
negativos = filter (< 0)    -- con el predicado (<0)
```

o a :

```haskell
noVacias :: [[a]] -> [[a]]
noVacias = filter (not . null) -- con ste predicado para seleccionar las listas no vac as
```