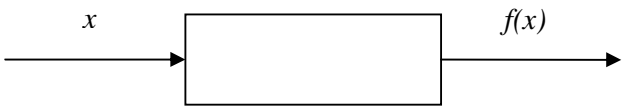
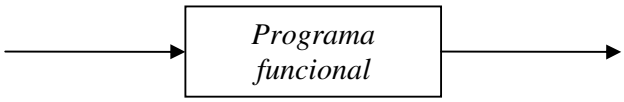
¿Qué es un programa en el paradigma funcional?

¿Y qué es una función?

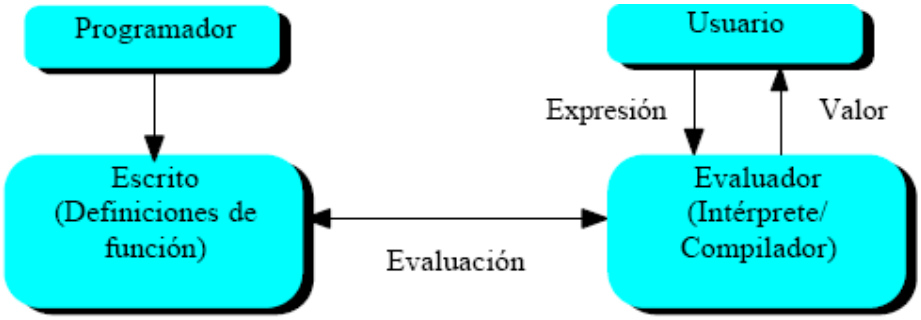


En el sentido matemático, una función implica que un hecho depende de otro.
F(x) = y = log x, implica que el valor de *y* depende del logaritmo *x* (está "en función de").

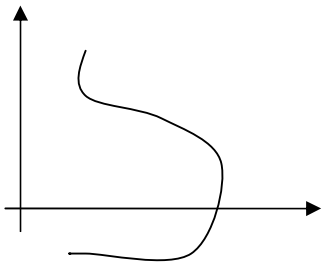
Entonces, un programa según el Paradigma Funcional será una función, una transformación de un input en un output…



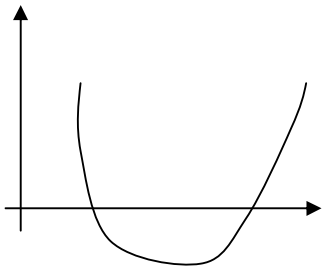
El output dependerá del input que se ingrese.
La metáfora asociada al paradigma funcional es la **calculadora**



Esto, ¿es una función?



No, porque para cada *x* sólo puede haber un *y* posible. Matemáticamente: para cada elemento del dominio puede haber una sola imagen:



Veamos un ejemplo en pseudocódigo:

```
Program Prueba;
var
   flag: BOOLEAN;

function f (n: INTEGER): INTEGER;
begin
   flag ← NOT flag;
   if flag then
      ↑ n;
   else
      ↑ 2 * n;
   endif
end;
```

Comentarios

↑	return / devuelve un valor
←	asignación de un valor o expresión a una variable

¿Qué pasa cuando pruebo?

```
--- Programa Principal
begin
   flag ← true;
   ...
   write( f(1) );    <- escribe 2
   write( f(1) );    <- escribe 1
   ...
   write( f(1) + f(2) );  <- escribe 4
   write( f(2) + f(1) );  <- escribe 5
   ...
end
```

Bueno, entonces f no es matemáticamente una función, para un mismo input tengo dos resultados posibles. Además, no es reflexiva ($f(1) \neq f(1)$) ni conmutativa ($f(1) + f(2) \neq f(2) + f(1)$).

Transparencia referencial

f(1) es una expresión *E* cuyo resultado da un valor *V*.

Decimos que una solución tiene transparencia referencial si podemos reemplazar en un programa todas las expresiones *E* por el valor *V* sin alterar el resultado del programa, independientemente del lugar donde aparezca *E* (no depende del contexto de ejecución, ni del orden de evaluación de dicha expresión).

O sea, si en cada lugar donde apareciera f(1) pudiéramos encontrar un *único valor* V que corresponde al resultado de evaluar f(1), entonces la solución tendría la propiedad de tener transparencia referencial (que justamente no es el caso, V puede ser 1 ó 2).

¿Qué ventaja tiene? Es mucho más sencillo demostrar la corrección de un programa (simplifica el testing).

¿Dónde está el problema que hace que mi solución no tenga transparencia referencial? Bien por el que dijo el flag: el valor de la función depende de un contexto que es “recordado”.

Efecto colateral

El efecto colateral[1] (side effect) es una modificación que sobrevive al bloque de código que la genera. Ojo, no es lo mismo que asignación destructiva, que es la simple modificación del valor contenido en una variable.

Ejemplo:

```
function contar_vocales(String palabra) : integer
begin
   var
      cantidad_vocales : integer;
      i : integer;

   cantidad_vocales ← 0;
   for i ← de 1 a LONGITUD(palabra)
   begin
      if (palabra(i) es vocal)
      begin
         cantidad_vocales ← cantidad_vocales + 1
      end
   end
   ↑ cantidad_vocales
end
```

Aquí vemos que hay una asignación destructiva de la variable cantidad_vocales (se pisan sucesivamente los valores), pero eso no afecta al resultado de la función, ya que el alcance de la variable es local[2]… En cambio la función f se ve afectada por cualquier modificación de la variable flag (porque flag es una variable de scope global) y en el ejemplo de f **sí** tenemos **efecto colateral**.

El paradigma funcional no tiene efecto colateral, no permite asignar en más de un contexto el valor de una variable. Es más, tengo el concepto matemático de variable muy distinto al de una celda de memoria.

Variables

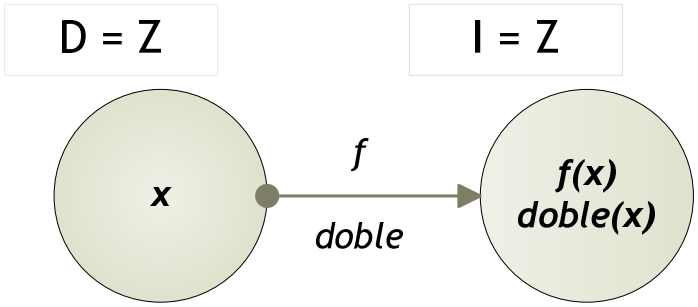
Variable (paradigma imperativo) = posición de memoria donde voy guardando valores. Me sirve para recordar estados intermedios[3].

Variable (paradigma funcional) = la variable dependiente y se calcula a partir de una variable independiente x. Por eso se suele definir una función acompañando la variable independiente: Velocidad = V(t). No existe la idea de tener x = x + 1. Si tengo x = 2 , x = 3 corresponden a dos puntos diferentes del dominio que tendrán sus correspondientes valores para la función. ¿Se entiende? No son posiciones de memoria, son incógnitas temporales que termino resolviendo cuando quiero conocer el valor de la función en un punto determinado.

Bueno, vamos con algunos ejemplitos para que no se aburran tanto…

Una función representa una relación entre un conjunto dominio y un conjunto imagen.
Ej.: `doble: Z->Z / doble(x) = 2 * x`

[1] A veces también llamado "efecto de lado" en viejas cursadas
[2] Lo mismo ocurre con la variable de iteración **i**
[3] Esta idea fue diseñada por el matemático húngaro Lajos Neuman (John Von Neumann)



en este caso la función 'doble' cuyo dominio son los enteros y cuyo conjunto imagen también son los enteros, está definida como el doble de cada elemento del dominio.

Para implementar esta abstracción utilizaremos el lenguaje Haskell. En un archivo de texto generamos la definición de la función doble:

```
doble :: Int -> Int
doble x = 2 * x
```

pdp.hs

Como vemos dentro de este lenguaje el signo = nos define una igualdad en el sentido más matemático posible.

```
f(x) = doble x = y = 2 * x
```

Desde el intérprete interactivo de Haskell cargamos el archivo de funciones predefinidas (en nuestro caso: pdp.hs):

```
Main> :a pdp.hs
```

Y luego evaluamos la expresión doble con el parámetro 4:

```
Main> doble 4
8
```

Entonces se reduce la expresión a un valor. Decimos que aplicamos el parámetro 4 a la función doble, la cual por tener un solo parámetro en su definición produjo su reducción.
Ahora bien, si alguien se fijó atentamente entre el mapeo entre la definición matemática y la definición haskell hay una particularidad que son los conjuntos de partida/llegada

en matemático era `Z->Z`
en Haskell `Int->Int`

esto quiere decir que desde Haskell vemos a los conjuntos que definen la función como a un tipo de dato. Por eso decimos que la función tiene un tipo y es parte característica de ella. En particular el tipo de doble es:

```
Main> :t doble
doble::Int->Int
```

Aclaración importante: no es necesario definir el tipo de mi función en Haskell dado que el motor sabe inferir éstos a partir de la implementación que nosotros hagamos. Por ende también es válido definir la función doble como:

```
doble x = 2 * x
```

2 * x ➔ el operador (*) tiene que poder aplicarse sobre 2 y sobre x. Como 2 es un número, está todo bien si x es también un número. Si x es un String no lo puedo multiplicar por dos, el operador (*) no es válido si el tipo de x no es el correcto. ¿Quién hace este chequeo? El compilador.

Tres tipos de chequeo que el compilador hace:

1) Si defino `doble x = "pepe" * x`: el valor "pepe" no es del tipo adecuado para el operador (*).

2) Si defino la función doble como:

```
doble :: String -> Int
```

no va a funcionar cuando quiera operar (*) con un argumento de tipo String:

```
doble x = 2 * x
```

3) Si pido evaluar la expresión
`Main> doble "2"`, el valor "2" (String) no coincide con el tipo Int.

Es por eso que Haskell es un lenguaje con chequeo estático de tipos.

Tener chequeo de tipos estático o dinámico es una característica del lenguaje y no del paradigma. Hay otros lenguajes funcionales como *Joy*[4] donde el compilador no chequea estrictamente los tipos de los argumentos y las expresiones, por eso es un lenguaje *con tipado dinámico*.

Ahora ¿qué pasa si quiero obtener el doble del doble de un número? Bueno lo más fácil es pensar en:

```
Main> doble ( doble 2 )
```

Básicamente lo que esta pasando es que la reducción de la expresión comienza de izquierda a derecha (excepto expresiones matemáticas) y el único elemento para forzar la precedencia que dispongo son los paréntesis.

Otros ejemplos de funciones simples:

```
celsiusToFahr c = c *  9/5  + 32
esMultiploDeTres n = rem n 3 == 0
```

Identificadores y Operadores

Hemos visto que las funciones tienen la notación del tipo:

```
nombreDeFuncion argumento1 argumento2 … argumentoN
```

[4] http://www.latrobe.edu.au/philosophy/phimvt/joy.html

y yo pido evaluar una función de esa misma manera. No obstante, Haskell admite dos maneras de nombrar una función:

- Mediante un identificador (o nombre de función como hemos visto en los casos doble, celsiusToFahr y esMultiploDeTres)
- Mediante un operador.

Los operadores se utilizan usando por default una notación infija, lo que facilita la lectura matemática:

```
4 + 3
x + y
```

También es posible usar el operador como si fuera un identificador encerrándolo entre paréntesis:

```
(+) 2 3
```

–este tipo de notación se llama prefija-

Otros operadores: ==, -, <, >, %, ^, etc.

Ejemplo de una función por partes (ecuaciones por guardas)

En matemática podíamos definir una función por partes (o definición "por trozos"). Para ciertos valores del dominio mi conjunto imagen era uno y para otros valores mi conjunto imagen era otro. Un ejemplo es la función max:

$$\text{max} : Z^2 \rightarrow Z \begin{cases} \text{max } (x,y) = x & x > y \\ \text{max } (x,y) = y & y >= x \end{cases}$$

$\max : ZxZ \rightarrow Z$

en haskell esto se traduce a:

```
max x y | x > y     = x
        | otherwise = y
```

notar el espacio de la segunda línea. Sintácticamente se debe dejar un espacio de indentación para el segundo pipe. La definición de min x y sería ? : ....

¿Qué otros conceptos de Análisis I recuerdan?

- **Composición de funciones:** doble (doble x), ¿no es lo mismo que (doble ∘ doble) x? Exacto. Ya lo conocen, vamos a profundizar.
- **Recursividad:** claro… el factorial nos va a venir al pelo. Y veremos que la recursividad siempre viene ligada a la inducción.
- **Límite:** mmm… quizás algún ejercicio.
- **Derivadas, Integrales:** ¿qué tipo de concepto representan? f '(x) expresa la razón de cambio de una función. Que termina siendo… ¡otra función! Eso tiene un nombre (una función que depende de la función, o sea una función de orden superior) y se va a ver.