# Apuntes Programación Lógica Funcional

## Estilos de Programación

(también llamados estándares de código o convención de código) es un término que describe convenciones para escribir código fuente en ciertos lenguajes de programación. El estilo de programación es frecuentemente dependiente del lenguaje de programación que se haya elegido para escribir. Por ejemplo el estilo del lenguaje de Programación C variará con respecto al del lenguaje BASIC.

### Características del Estilo

Una pieza clave para un buen estilo es la elección apropiada de nombres de variable. Variables pobremente nombradas dificultan la lectura del código fuente y su comprensión.

Como ejemplo, considérese el siguiente extracto de pseudocódigo:

```
get a b c

if a < 24 and b < 60 and c < 60

  return true

else

  return false
```

Debido a la elección de nombres de variable, es difícil darse cuenta de la función del código. Compárese ahora con la siguiente versión:

```
 get horas minutos segundos

 if horas < 24 and minutos < 60 and segundos < 60

   return true

 else
```

```
    return false
```

La intención el código es ahora más sencilla de discernir, "dado una hora en 24 horas, se devolverá true si es válida y false si no".

## Nombres de Variable Apropiadas.

Una piedra clave para un buen estilo es la elección apropiada de nombres de variable. Variables pobremente nombradas dificultan la lectura del código fuente y su comprensión. y Como ejemplo, considérese el siguiente extracto de pseudocódigo:

```
get a b c

if a < 24 and b < 60 and c < 60

  return true

else

  return false
```

Debido a la elección de nombres de variable, es difícil darse cuenta de la función del código. Compárese ahora con la siguiente versión:

```
 get horas minutos segundos

 if horas < 24 and minutos < 60 and segundos < 60

   return true

 else

   return false
```

La intención el código es ahora más sencilla de discernir, "dado una hora en 24 horas, se devolverá true si es válida y false si no".

## Estilo de indentación

Estilo de indentación, en lenguajes de programación que usan llaves para indentar o delimitar bloques lógicos de código, como por ejemplo C, es también un punto clave el buen estilo. Usando un estilo lógico y consistente hace el código de uno más legible. Compárese:

```
if(horas < 24 && minutos < 60 && segundos < 60){

   return true;

}else{

   return false;

}
```

o bien:

```
if(horas < 24 && minutos < 60 && segundos < 60)

{

   return true;

}

else

{

   return false;

}
```

con algo como:

```
if(horas<24&&minutos<60&&segundos<60){return true;}
```

```
else{return false;}
```

Los primeros dos ejemplos son mucho más fáciles de leer porque están bien indentados, y los bloques lógicos de código se agrupan y se representan juntos de forma más clara.

Valores booleanos en estructuras de decisión Algunos programadores piensan que las estructuras de decisión como las anteriores, donde el resultado de la decisión es meramente una computación de un valor booleano, son demasiado prolijos e incluso propensos al error. Prefieren hacer la decisión en la computación por sí mismo, como esto:

```
return horas < 12 && minutos < 60 && segundos < 60;
```

La diferencia es, con frecuencia, puramente estilística y sintáctica, ya que los compiladores modernos producirán código objeto idéntico en las dos formas.

## Bucles y estructuras de control

El uso de estructuras de control lógicas para bucles también es parte de un buen estilo de programación. Ayuda a alguien que esté leyendo el código a entender la secuencia de ejecución (en programación imperativa). Por ejemplo, el siguiente pseudocódigo:

```
cuenta = 0

while cuenta < 5

  print cuenta * 2

  cuenta = cuenta + 1

endwhile
```

El extracto anterior cumple con las dos recomendaciones de estilo anteriores, pero el siguiente uso de la construcción for hace el código mucho más fácil de leer:

```
for cuenta = 0, cuenta < 5, cuenta=cuenta+1

  print cuenta * 2
```

En muchos lenguajes, el patrón frecuentemente usado "por cada elemento en un rango" puede ser acortado a:

```
for cuenta = 0 to 5

  print cuenta * 2
```

## Espaciado

Los lenguajes de formato libre ignoran frecuentemente los espacios en blanco. El buen uso del espaciado en la disposición del código de uno es, por tanto, considerado un buen estilo de programación.

Compárese el siguiente extracto de código C:

```
int cuenta; for(cuenta=0;cuenta<10;cuenta++)

{printf("%d",cuenta*cuenta+cuenta);}
```

con:

```
int cuenta;

for (cuenta = 0; cuenta < 10; cuenta++)

{

   printf("%d", cuenta * cuenta + cuenta);

}
```

En los lenguajes de programación de la familia C se recomienda también evitar el uso de caracteres tabulador en medio de una línea, ya que diferentes editores de textos muestran su anchura de forma diferente.

El lenguaje de programación Python usa indentación para indicar estructuras de control, por tanto se requiere obligatoriamente una buena indentación. Haciendo esto, la necesidad de marcar con llaves ({ y }) es eliminada, y la legibilidad es mejorada sin interferir con los estilos de codificación comunes.

Con todo, esto lleva frecuentemente a problemas donde el código es copiado y pegado dentro de un programa Python, requiriendo un tedioso reformateado. Adicionalmente, el código Python se vuelve inusable cuando es publicado en un foro o página web que elimine el espacio en blanco.

### Evaluación de expresiones

En general, salvo que se relacionen con las mencionadas sentencias modificadoras del flujo, las palabras-clave señalan al compilador aspectos complementarios que no alteran el orden de ejecución dentro de la propia sentencia. Este orden viene determinado por cuatro condicionantes:

1. Presencia de paréntesis que obligan a un orden de evaluación específico.
2. Naturaleza de los operadores involucrados en la expresión (asociatividad).
3. Orden en que están colocados (precedencia).
4. Providencias (impredecibles) del compilador relativas a la optimización del código.

## Paradigma de Programación

Un paradigma de programación es una propuesta tecnológica adoptada por una comunidad de programadores y desarrolladores cuyo núcleo central es incuestionable en cuanto que únicamente trata de resolver uno o varios problemas claramente delimitados; la resolución de estos problemas debe suponer consecuentemente un avance significativo en al menos un parámetro que afecte a la ingeniería de software.

## Tipos más comunes de paradigmas de programación

### Programación imperativa o por procedimientos:

Es el más usado en general, se basa en dar instrucciones al ordenador de como hacer las cosas en forma de algoritmos. La programación imperativa es la más usada y la más antigua, el ejemplo principal es el lenguaje de máquina. Ejemplos de lenguajes puros de este paradigma serían el C, BASIC o Pascal.

### Programación orientada a objetos:

Está basada en el imperativo, pero encapsula elementos denominados objetos que incluyen tanto variables como funciones. Está representado por C++, C#, Java o Python entre otros, pero el más representativo sería el Smalltalk que está completamente orientado a objetos.

### Programación dinámica:

está definida como el proceso de romper problemas en partes pequeñas para analizarlos y resolverlos de forma lo más cercana al óptimo, busca resolver problemas en O(n) sin usar por tanto métodos recursivos. Este paradigma está más basado en el modo de realizar los algoritmos, por lo que se puede usar con cualquier lenguaje imperativo.

### Programación dirigida por eventos:

la programación dirigida por eventos es un paradigma de programación en el que tanto la estructura como la ejecución de los programas van determinados por los sucesos que ocurran en el sistema, definidos por el usuario o que ellos mismos provoquen.

### Programación declarativa:

está basado en describir el problema declarando propiedades y reglas que deben cumplirse, en lugar de instrucciones. Hay lenguajes para la programación funcional, la programación lógica, o la combinación lógico-funcional. Unos de los primeros lenguajes funcionales fueron Lisp y Prolog.

## Programación funcional:

La programación funcional es un paradigma de programación declarativa basado en el uso de verdaderas funciones matemáticas. En este estilo de programación las funciones son ciudadanas de primera clase, porque sus expresiones pueden ser asignadas a variables como se haría con cualquier otro valor; además de que pueden crearse funciones de orden superior.

La programación funcional tiene sus raíces en el cálculo lambda, un sistema formal desarrollado en los años 1930 para investigar la naturaleza de las funciones, la naturaleza de la computabilidad y su relación con la recursión. Los lenguajes funcionales priorizan el uso de recursividad y aplicación de funciones de orden superior para resolver problemas que en otros lenguajes se resolverían mediante estructuras de control (por ejemplo, ciclos). Algunos lenguajes funcionales también buscan eliminar la mutabilidad o efectos secundarios; en contraste con la programación imperativa, que se basa en los cambios de estado mediante la mutación de variables. Esto significa que, en programación funcional pura, dos o más expresiones sintácticas idénticas (por ejemplo, dos llamadas a rutinas o dos evaluaciones) siempre devolverán el mismo resultado. Es decir, se tiene transparencia referencial. Lo anterior también puede ser aprovechado para diseñar estrategias de evaluación que eviten repetir el cómputo de expresiones antes vistas, ahorrando tiempo de ejecución.

Los lenguajes de programación funcional, especialmente los puramente funcionales, han sido enfatizados en el ambiente académico y no tanto en el desarrollo comercial o industrial. Sin embargo, lenguajes de programación funcional como Lisp (Scheme, Common Lisp, etc.), Erlang, Rust, Objective Caml, Scala, F# y Haskell, han sido utilizados en aplicaciones comerciales e industriales por muchas organizaciones. También es utilizada en la industria a través de lenguajes de dominio específico como R (estadística), Mathematica (cómputo simbólico), J y K (análisis financiero). Los lenguajes de uso específico usados comúnmente como SQL y Lex/Yacc, utilizan algunos elementos de programación funcional, especialmente al procesar valores mutables. Las hojas de cálculo también pueden ser consideradas lenguajes de programación funcional.

## Programación lógica:

basado en la definición de relaciones lógicas, está representado por Prolog.

### Programación con restricciones:

similar a la lógica usando ecuaciones. Casi todos los lenguajes son variantes del Prolog.

### Programación multiparadigma:

es el uso de dos o más paradigmas dentro de un programa. El lenguaje Lisp se considera multiparadigma. Al igual que Python, que es orientado a objetos, reflexivo, imperativo y funcional.1

### Lenguaje específico del dominio o DSL:

se denomina así a los lenguajes desarrollados para resolver un problema específico, pudiendo entrar dentro de cualquier grupo anterior. El más representativo sería SQL para el manejo de las bases de datos, de tipo declarativo, pero los hay imperativos, como el Logo.

## Programación Funcional

La programación funcional, o mejor dicho, los lenguajes de programación funcionales, son aquellos lenguajes donde las variables no tienen estado — no hay cambios en éstas a lo largo del tiempo — y son inmutables — no pueden cambiarse los valores a lo largo de la ejecución. Además los programas se estructuran componiendo expresiones que se evalúan como funciones. Dentro de los lenguajes funcionales tenemos Lisp, Scheme, Clojure, Haskell, OCaml y Standard ML, entre otros. Estos lenguajes están diversidad de tipificación, donde se encuentran lenguajes dinámicos, estáticos y estáticos fuertes.

En los lenguajes funcionales las instrucciones cíclicas como for, while y do-while no existen. Todo se procesa usando recursividad y funciones de alto orden.

Esto se debe a los fundamentos matemáticos de la mayoría de los lenguajes funcionales, principalmente con bases en el sistema formal diseñado por Alonzo

Church para definir cómputos y estudiar las aplicaciones de las funciones llamado Cálculo Lambda. En este sistema formal se puede expresar recursividad en las funciones, y entre otras cosas interesantes, se pueden expresar combinadores — funciones sin variables libres — como el Combinador de Punto Fijo o Y-Combinator, que expresa recursividad sin hacer llamadas recursivas.

En el Cálculo Lambda existen tres transformaciones esenciales, la conversión αα, la reducción ββ y la conversión ηη. En la conversión α se sustituyen los nombres de las variables para dar mas claridad a la aplicación de las funciones, por ejemplo evitando duplicados en sus nombres. En la reducción ββ se traza el llamado de las funciones sustituyendo las funciones por sus expresiones resultantes.
Finalmente en las conversiones η se busca las equivalencias de trazado de funciones sustituyéndolas por sus equivalentes. Estas transformaciones también pueden ser aplicadas en los lenguajes funcionales — o en su mayoría — dando lugar lenguajes que cuentan con una gran expresividad y consistencia.

Les pondré el clásico ejemplo del chiste geek del castigo “Debo poner atención en clases”. La respuesta geek expresada en PHP esta escrita a continuación. Donde PHP es un lenguaje dinámico, no necesita declarar variables y es un lenguaje orientado a objetos con raíces imperativas. Sus instrucciones son paso a paso, y no constituyen una única expresión reducible.

```php
<?php

    /* codigo PHP */

    for ($i = 0; $i < 500; $i++) {

        echo "Debo poner atención en clase";

    }

 ?>
```

Si usamos Haskell como ejemplo, que es un lenguaje funcional con tipificación estática fuerte, requiere que las variables sean declaradas con un tipo — la mayoría de las veces — y es muy expresivo, donde el siguiente ejemplo dice repetir la

cadena, tomar 500 elementos y con esa lista ejecutar la función monádica putStrLn, que esta hecha para el Monad IO e imprime el mensaje las 500 veces solicitada.

```
module Main (main) where



-- codigo Haskell



main :: IO ()

main = mapM_ putStrLn $ take 500 $ repeat "Debo poner atención"
```

En Lisp sería similar, pero Lisp es de tipificación dinámica y no necesita declarar variables, dando lugar a un programa muy simple de una sola linea. Donde también tenemos lenguajes como Clojure, que es un dialecto de Lisp y soporta construcciones muy similares a las del ejemplo en Lisp, dando lugar a programas expresivos y simples, pero que corren sobre la máquina virtual de Java o JVM.

```
;;; codigo Lisp



(loop repeat 500 do (format t "Debo poner atencion en clases~%"))
```

Un ejemplo clásico para la conversión η en Haskell, es reducir las llamadas a funciones en su combinador de identidad. Por ejemplo se tiene la función $f(g(x))\diamond(\diamond(\diamond))$, que en Cálculo Lambda se expresa como $\lambda x.(\lambda y.y)x\lambda\diamond.(\lambda\diamond.\diamond)\diamond$, se puede reducir a $g(x)\diamond(\diamond)$, que se expresa como $\lambda y.y\lambda\diamond.\diamond$ en Cálculo Lambda. Esto expresado en Haskell, se vería como el siguiente ejemplo, donde absN y absN' son funciones equivalentes y absN' es la reducción ηη de absN.

```
absN :: Num a => a -> a

absN n = abs n
```

```
absN' :: Num a => a -> a

absN' = abs
```

Actualmente los lenguajes orientados a objetos más comunes están integrando características funcionales, como Java, que acaba de incluir funciones anonimas. Pero también están los lenguajes que a lo largo de su historia han sido multi-paradigma, como Python, e implementa características funcionales, procedurales y orientadas a objetos. El bien conocido algoritmo para verificar si un RUT es válido o no, se puede expresar funcionalmente en Python como esta escrito en el siguiente ejemplo.

```
def val_rut(rut):

    """

    Valida un string con un RUT con el guion incluido, retornando

    cero si es valido.



    Ejemplo: print(val_rut("22222222-2"))

    """

    return cmp(rut[-1],

               str((range(10) + ['K'])[

                   (11 - sum(map(lambda x: (int(x[0]) * x[1]),

                                 zip(reversed(rut[:-2]),

                                     (2 * range(2, 8))))) % 11)]))
```

Como se aprecia en el ejemplo, la validación se realiza utilizando expresiones o llamadas a funciones, sin uso de variables con estado y mutabilidad, donde cada llamada a una función se puede reducir a un valor determinado, y como resultado final se tiene un valor cero o distinto de cero que indica si el RUT es válido. Este mismo algoritmo funcional, se puede expresar en Haskell con llamadas muy similares, debido a que los nombres de las funciones y funciones de alto orden son bastante comunes entre los lenguajes funcionales.

```
valRut :: String -> Bool

valRut s = ((['0'..'9'] ++ ['K'])

             !! (11 - sum(zipWith (*)

                          (fmap digitToInt $ drop 2 $ reverse s)

                          (take 10 $ cycle [2..7])) `mod` 11)) == (last s))
```

De estos dos ejemplos, se puede decir que son funciones puras, principalmente debido a que no tienen variables libres y son una única expresión sin estado y no mutable a lo largo de la ejecución. El problema de la pureza es conceptualmente algo que se idealiza en la programación funcional, siendo abordado de diferentes formas por diferentes lenguajes. El objetivo es mantener las funciones y rutinas puras. En Haskell, con su abstracción más clásica conocida con el nombre de Mónada, permite entregar pureza a expresiones que parecen no ser puras, y en términos muy sencillos el Mónada reúne una identidad y una composición de funciones del tipo $f(g(x))$ (), todo a través de un tipo de dato que permite componer funciones sin abandonar ese tipo de dato y darle un aspecto procedural.