## UNIDAD 3 – PROGRAMACIÓN DECLARATIVA

*"Quien pregunta parece ignorante durante un minuto. Quien no pregunta permanece ignorante el resto de su vida".*

*Proverbio Chino*

### Introducción

La Programación Clásica o Imperativa no es la única forma de escribir programas. Su enfoque, también conocido como operacional, se basa en dar órdenes a la computadora. Podríamos hablar de un punto de vista activo (cómo resolver el problema). A partir de la definición del problema el programador desarrolla un algoritmo o método para resolverlo. Desde este punto de vista se entiende el universo como una entidad dinámica (que cambia de estado con el tiempo), y el programa como un agente modificador que altera dicho estado.

Las computadoras actuales responden al enfoque imperativo, puesto que poseen una memoria (que almacena el estado del "universo") y un microprocesador que procesa secuencias de órdenes. El formalismo matemático que modeliza esta filosofía son los Autómatas Finitos.

La Programación Declarativa por el contrario intenta centrarse en la descripción formal del problema (qué queremos hacer), y no en la secuencia de pasos para resolverlo. Al eliminar el concepto de secuencia temporal de pasos desaparece el concepto de estado de la máquina, el tiempo y por lo tanto el concepto de asignación (que es la operación imperativa por antonomasia). Esta ausencia de asignación y de estado de la máquina (y por tanto de memoria) se denomina transparencia referencial.

No existe, por tanto, el concepto de asignación a:=b, sólo el concepto matemático de igualdad. Las variables las entenderemos como incógnitas, cuyo valor puede ser o no conocido, pero nunca modificado.

Todo esto es una filosofía, una forma de entender los problemas. Distintas filosofías permitirán resolver los problemas de diferentes formas, pero las soluciones serán equivalentes (puesto que todos los enfoques de la computación lo son). Lo que se puede resolver mediante un enfoque se puede resolver mediante cualquiera de los otros. De aquí surge el concepto de computabilidad: un problema es computable si existe algún mecanismo automático para resolverlo, y si existe una solución dentro de uno de los enfoques, existen soluciones equivalentes en el otro.

Normalmente los enfoques se mezclan y es una cuestión de filosofía de programación el aplicarlos. Habitualmente resolveremos los problemas empleando la aproximación que proporcione una solución más sencilla. Puesto que los ordenadores responden al enfoque imperativo, en la práctica no existen lenguajes 100% declarativos. En general, todos los lenguajes de programación aplican conceptos de los dos enfoques fundamentales.

La Programación Declarativa nos permitirá, pues, enfocar los problemas desde puntos de vista diferentes a los de la Programación Imperativa, lo que en muchos casos facilitará encontrar soluciones mejores a dichos problemas. A su vez, este estilo de programación se subdivide en otros dos: la Programación Funcional y la Programación Lógica.

### Un poco de historia

La programación es una actividad muy formal, es un lugar común: "las computadoras hacen exactamente lo que se les diga que hagan, por eso es necesario expresarlo con precisión", desafortunadamente la mayoría de los programadores no lo ven así y construyen sus programas pensando como le ordenarían a una persona lo que debe computar con la precisión suficiente para que no requiera usar su ingenio. La razón de que esto funciona en la práctica es por lo relativamente fácil de visualizar de esta manera antropomórfica.

En un principio se pensaban los programas dibujando diagramas de flujo que se traducían a algún lenguaje de programación como Fortran, Basic, o Cobol. Estos lenguajes en ese momento no tenían estructuras de control, if-then-else, while-do, y es bastante conocido lo difícil que es entender un programa hecho usando libremente la instrucción GoTo, fue a partir del lenguaje Algol que se incorporaron las estructuras de control, y los bloques begin-end.

El siguiente paso para poner más orden en la programación fue usar estructuras de control para la decisión y la iteración, así como un mejor uso de la abstracción procedural, esto es el uso de subprogramas (call). Este enfoque se basó en el análisis de lo general a lo particular en inglés Top-Down que consiste en dividir en partes más sencillas lo que se percibe como una unidad.

La programación estructurada es muy útil para una muy amplia clase de problemas, los proyectos pequeños llevados a cabo por un solo programador, o cuando se pretende resolver un problema en particular como es el caso de la solución de ecuaciones o sistemas de ecuaciones. El lenguaje Pascal es tal vez más representativo de este periodo.

Después paso un breve período en que llamó mucho la atención la Inteligencia Artificial en donde los lenguajes predominantes eran Lisp y Prolog, y se cuenta con enfoques muy distintos a los sistemas más tradicionales como empleados para hacer cálculos o los que se usan en ambientes administrativos.

Afortunadamente el oportunismo de los publicistas duró muy poco con la Inteligencia Artificial y se aprovecharon y tal vez hasta el momento de la programación orientada a objetos. Que surgió como una evolución natural de la programación estructurada con antecedentes en el lenguaje Simula de mediado de los 1960s. El enfoque en que se basan es en aislar del detalle interno cada una de las partes que integran el sistema dando una interfaz definiendo las operaciones que puede hacer el objeto. La programación dejó de ser monolítica para poder reutilizar y mantener mejor el código.

La idea de la programación orientada a objetos fue ampliamente aceptada en parte por la analogía que se hizo de los objetos con los chips. Antes para diseñar una computadora o cualquier dispositivo electrónico por ejemplo un radio, se tenía que diseñar el receptor, los amplificadores, el ecualizador, la fuente de poder, y las demás partes, partiendo de transistores, diodos, resistencias, capacidades, bobinas, y demás componentes, lo que es muy complicado, en cambio con los chips se busca en el catálogo el receptor más adecuado, los amplificadores más adecuados, la fuente de poder, etc. Y quien diseña el dispositivo solo se preocupa por integrar correctamente las partes, y puede confiar en que cada una hace lo que dice su especificación.

La dificultad del paradigma de orientación a objetos estriba en saber cuales son los objetos básicos y en encontrar las jerarquías correctas.

Aunque desde el punto de vista de la mercadotecnia lleva unos veinte años de éxito, lo que realmente sucede es que aunque se usan lenguajes con características de orientación a objetos la mayoría de los programadores siguen razonando acerca de los programas con el paradigma de la programación estructurada, aunque en los últimos cinco años se han comenzado a colocar en el mercado productos que ayudan a organizar el código, estos se basan en una metáfora antropomórfica que solo nos permite hacer una representación del mundo real como presuponemos que es, basada en la percepción particular del analista, que no necesariamente se asemeja a la realidad, cada individuo tiene una historia distinta y distintas habilidades en sus sentidos, por lo que cada persona percibe distinto el mundo, aunque hay partes comunes que forman parte de la cultura y que permiten la comunicación y el entendimiento entre individuos.

La *programación funcional* **no** representa un siguiente paso en la evolución. Tal vez por esto a los programadores imperativos se les dificulta más comprenderla, que a los “inexpertos”.

El origen de la programación funcional esta en las matemáticas, en las más abstractas y que muchos piensan que no son de utilidad práctica, de hecho se acostumbra llamarles Matemáticas Puras, en contraposición a las Matemáticas Aplicadas.

La Informática y las Ciencias de la Computación, como las conocemos ahora, vienen del

trabajo de importantes Matemáticos, y Filósofos de mediados del siglo XIX.

A diferencia de la programación imperativa, la programación funcional en su estado actual nos permite organizar y desarrollar "programas" de manera similar a como se hace en las Matemáticas por lo que estrictamente hablando no es algo novedoso, más bien es bastante antiguo y su eficacia esta plenamente demostrada.

El problema para su divulgación es el mismo que el de las Matemáticas, y es que el aprendizaje es constructivo, si alguno de los conceptos no se comprende es difícil pasar a otros más complejos que se derivan de los básicos.

La estrategia seguida en la programación funcional es definir formalmente la semántica, generalmente se usan constructores y se hace de manera axiomática, las definiciones se pueden agrupar en módulos, y como todos los objetos son "ciudadanos de primera clase" tanto los datos como las funciones pueden pasarse o ser el resultado de la evaluación de una función.

Aunque no es algo nuevo pues existe ya algo de trabajo hecho, en un futuro tal vez no muy lejano, será más popular la construcción de especificaciones formales, y se publicarán como se publican las Matemáticas. Y como actualmente se pretende con la construcción de bibliotecas de objetos.

Pero para que suceda lo anterior, no es necesario esperar a que cambien los planes de estudios en todas las universidades para cambiar su enfoque, esto es algo que se ha ido incorporando gradualmente añadiendo materias como Lógica, y Matemáticas Discretas a las currículas de las carreras relacionadas con la Informática.

Por lo anterior parece importante introducir al tema a quienes han sido formados como programadores imperativos.

**Paradigmas de Programación**

Definiciones:

- ***"Es un modelo básico de diseño y desarrollo de programas"***
- ***"Es una colección de modelos conceptuales que juntos modelan el proceso de diseño y determinan, al final, la estructura de un programa"***

Un paradigma de Programación engloba a ciertos lenguajes que comparten:

- **Elementos estructurales:**
  - ¿con qué se confeccionan los programas?
- **Elementos metodológicos:**
  - ¿cómo se confecciona un programa?

Los paradigmas de programación pueden clasificarse en dos grandes categorías, los procedimentales u operacionales y los declarativos. Los primeros se distinguen por que cuentan con estructuras de control, con una semántica que se basa en la transición de estados, esto quiere decir que para obtener el resultado del cómputo cada instrucción que se ejecuta cambia el estado de una máquina hasta que se detiene la ejecución. Por otra parte los declarativos con una semántica de reducción, en este caso la entrada se reescribe aplicando las reglas almacenadas en la máquina hasta que ya no se puede aplicar ninguna regla, la máquina cuenta con un algoritmo que controla este proceso, por lo que solo es necesario declarar las reglas.

Resumiendo, tenemos:

- **Procedimentales u operacionales:**
  - Describen etapa a etapa el modo de construir la solución

- **Declarativos:**
  - Señalan las características que debe tener la solución, sin describir cómo procesarla.

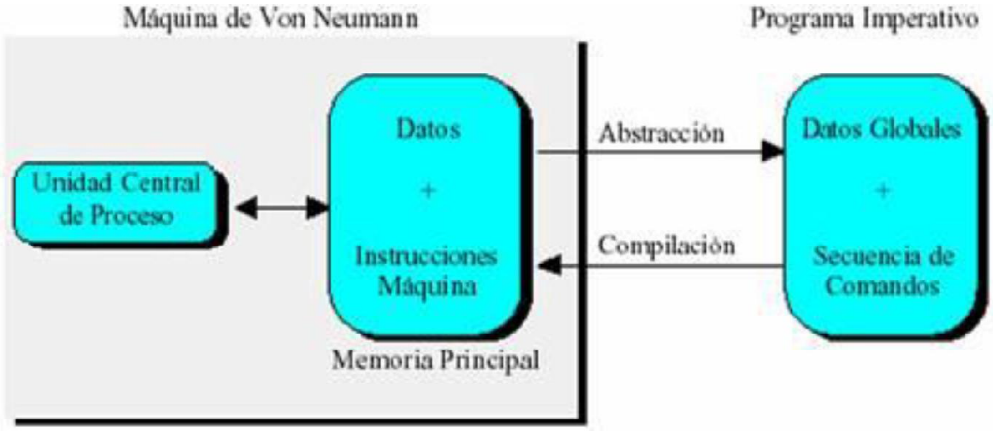
**Paradigmas Procedimentales u Operacionales**

Se dividen en dos grupos:

- **Programación Imperativa**
- **Programación Orientada a Objetos**

**La Programación Imperativa**

Es la más antigua, está basada en la máquina de Von Newman




- Está orientada a la arquitectura del computador
- No está orientada al modo de pensar humano, sino de la máquina
- El concepto básico es la instrucción o comando
  - Poseen asignaciones, saltos condicionales e incondicionales, bucles, etc.
  - Afectan o modifican el estado

Características fundamentales de este tipo de paradigma

- Concepto de celda de memoria (variable) para almacenar valores
- Operaciones de asignación
- Repetición

En otras palabras, con los lenguajes imperativos se tiene que expresar el control, es decir **cómo** se va a realizar el proceso de cómputo. En los lenguajes imperativos, como su nombre lo dice se dan órdenes como:

*Asignación:*

**Contador :=10; /* pone un "10" en la localidad de memoria "Contador" */**

*Asignación:*

**x := 0; /* inicia la localidad de memoria "x" con "0" */**

*e iteración:*

```
repeat /*repite el siguiente bloque de instrucciones hasta que "Contador" tenga el valor "0"*/
{
...
}
until Contador = 0;
```

Nótese que en este ejemplo no se usa el término variable, sino localidad de memoria. Y es que en los lenguajes de programación convencionales el concepto de variable no es el mismo que en las matemáticas. Dijkstra utiliza el término más apropiado de constantes mutables.

***Ejemplo Números Primos***

```
Programa primos(input, output)
Constante n=50
Variables       i,,j : enteras
                iprimo : lógica
Inicio
        Para i desde 2 hasta n hacer
                (* ¿ Es primo y ? *)
                j =2;
                iprimo = verdaderp
                Mientras iprimo y (j<=i div 2) hacer
                        si ( (y mod j) <>0) entonces
                                j=j+1
                        sino
                                iprimo=falso
                        fin_si
                fin_mientras
                (* Si es primo imprime su valor *)
                si primo entonces
                        escribir (y)
                fin_si
        fin_para
Fin.
```

**La programación declarativa**

En los lenguajes declarativos solamente tenemos que expresar **qué** se quiere computar y no **cómo**. En los lenguajes declarativos se utilizan variables en el sentido en que lo hace la Matemática.

Compare los dos programas el primero imperativo y el segundo declarativo.

*Versión Imperativa:*

```
suma(Contador)
        X := 0;
        repetir
                X := Contador + X
                Contador := Contador - 1;
        hasta que Contador = 0;
devolver(X);
```

*Versión Declarativa:*

```
suma(0) = 0
suma(X) = X + suma(X - 1)
```

El primer programa consta de una máquina con dos localidades de memoria una llamada "X", y la otra llamada "Contador". El estado inicial de la máquina esta dado por el valor que se le pasa a "contador", y la localidad "x" puede contener cualquier valor. El estado final del cómputo se alcanza cuando se cumple la condición: "Contador = 0", y el resultado se almacenará en "X". Si se efectúa el llamado suma(5) , los estados intermedios del cómputo serán los siguientes:

| Instrucción | ¿Contador = 0? | Contador | X |
|---|---|---|---|
| Suma(5) | Falso | 5 | Indefinido |
| X:=0 | Falso | 5 | 0 |
| X := Contador + X | Falso | 5 | 5 |
| Contador := Contador – 1 | Falso | 4 | 5 |
| X := Contador + X | Falso | 4 | 9 |
| Contador := Contador – 1 | Falso | 3 | 9 |
| X := Contador + X | Falso | 3 | 12 |
| Contador := Contador – 1 | Falso | 2 | 12 |
| X := Contador + X | Falso | 2 | 14 |
| Contador := Contador – 1 | Falso | 1 | 14 |
| X := Contador + X | Falso | 1 | 15 |
| Contador := Contador – 1 | Verdadero | 0 | 15 |

Lo anterior se conoce como una prueba de escritorio, y algunos programadores la usan para verificar sus programas. Cuando algo falla se puede hacer una prueba de escritorio no necesariamente con papel y lápiz, hay depuradores de programas que permiten ver el contenido de las localidades de memoria mientras se ejecuta el programa, este se puede ejecutar paso a paso o marcar los puntos de interés para que se detenga en éstos la ejecución.

Por otra parte la versión declarativa se obtiene reescribiendo la entrada hasta que ya no hay nada que reescribir.

En el caso del ejemplo anterior, suma(5), tenemos el siguiente cómputo:

```
Suma(5)                         => 5+Suma(5–1)                  =>
5+Suma(4)                       => 5+(4+Suma(4-1))              =>
5+(4+Suma(3))                   => 5+(4+(3+Suma(3-1)))          =>
5+(4+(3+Suma(2)))               => 5+(4+(3+(2+Suma(2-1))))      =>
5+(4+(3+(2+Suma(1))))           => 5+(4+(3+(2+(1+Suma(1-1)))))  =>
5+(4+(3+(2+(1+Suma(0)))))       => 5+(4+(3+(2+(1+0))))          =>
5+(4+(3+(2+1)))                 => 5+(4+(3+3))                  =>
5+(4+6)                         => 5+10                         =>
15
```

Como puede ver es el mismo mecanismo que se usa en el álgebra que se estudia en secundaria, no es necesario decir como se efectuará el cómputo, y analizando las fórmulas se puede analizar si el programa es correcto. En este ejemplo se restringe la entrada a enteros positivos y cero,

si se permite la misma definición incluyendo los negativos es fácil ver que suma(-1) nunca se detendrá. Por eso es necesario definir a que conjunto pertenecen las variables.

Para entender la diferencia entre los lenguajes declarativos, y los que no lo son tenemos que poner atención en que forma se expresan los algoritmos en el lenguaje en cuestión.

Antes que nada, un algoritmo se define como un conjunto de reglas que se aplican para obtener un resultado, es decir que en algún momento se llega a un resultado, es efectivo.

Un algoritmo consta de dos partes una Lógica que expresa QUE se conoce y una parte de Control que expresa COMO se resolverá el problema en cuestión, Kowalski utiliza la fórmula **Algoritmo=Lógica+Control** para destacar la ventaja de usar un lenguaje declarativo en donde el control forma parte del interprete o compilador, permitiendo expresar la Lógica del programa sin preocuparse por el control. No sucede así con los lenguajes imperativos donde no hay una separación clara de la lógica y el control. Quienes utilizan especificaciones formales mediante aseveraciones, precondiciones, y postcondiciones para verificar los programas imperativos, distinguen entre la Lógica y el Control, pero se encargan totalmente del control.

Cuando se trata por separado la lógica del control, es posible hacer optimizaciones sin alterar la componente Lógica del algoritmo, ya sea cambiando la máquina que interpreta un programa declarativo, o transformando las instrucciones de un programa imperativo, incluso les llaman transformadores de predicados obteniéndose así una versión más refinada del algoritmo siendo su Lógica idéntica al anterior, pero efectuando el mismo cómputo.

También es posible usar lenguajes declarativos para especificar formalmente el programa, y posteriormente transformarlo en uno imperativo, de manera similar a la que se usa en la verificación de programas imperativos.

Examinando el ejemplo de la suma de naturales, se puede ver que es más fácil de entender mejor la versión declarativa que la imperativa ya que para entender la imperativa es necesario simular la ejecución del programa y ver que cambio tienen las constantes mutables “X” y “Contador” para entender en que estado terminan y así comprender cual es el cómputo que efectúa el algoritmo, en cambio la versión declarativa requiere menos esfuerzo ya que no se tiene que simular la ejecución del algoritmo para entenderlo.

**¿Qué es la “Transparencia Referencial”?**

Es reemplazar una subexpresión por otra subexpresión igual que no altera el valor de una expresión.

Sea la expresión x = f(a) entonces podemos sustituir
x+x por f(a) + f(a), o incluso por 2 * f(a)

Características:

- Podemos reemplazar “iguales por iguales”. Esta propiedad **no** la cumplen los lenguajes imperativos.
- Permite razonar formalmente acerca de los programas
  - Razonamiento Ecuacional
  - Inducción Matemática
- Simplifica el desarrollo y depuración del software

La programación imperativa carece de la Transparencia referencial, en cambio la declarativa sí la posee.

**Ejemplo**

```
Program Prueba;
VAR
  flag:BOOLEAN;

FUNCTION f (n:INTEGER):
INTEGER;
BEGIN
     flag := NOT flag;
     IF flag THEN f := n;
             ELSE f := 2*n
END;
```

```
--- Programa Principal
BEGIN
 flag:=TRUE;
 ...
 WRITE( f(1) );
 WRITE( f(1) );
 ...
 WRITE( f(1) + f(2) );
 WRITE( f(2) + f(1) );
 ...
END.
```

```
<- Escribe 2
<- Escribe 1

<- Escribe 4
<- Escribe 5
```

No se cumplen propiedades matemáticas simples como la propiedad conmutativa, en el ejemplo, f(1) + f(2) no es igual a f(2) + f(1).

**Ventajas e Inconvenientes de la Programación Declarativa**

- ✓ La principal es que son independientes de la máquina y, como se ha comentado, referencialmente transparentes.
- ✓ La cantidad de código que debemos escribir es menor, aunque la representación formal de estos problemas puede resultar, en ocasiones, poco evidente.
- ✓ Podemos desentendernos del control. Aunque aquí existe una limitación: no podemos hacerlo totalmente.
- ✓ Los datos pueden ser tanto de entrada como de salida y se pueden utilizar datos parcialmente construidos, es decir, podemos empezar a ejecutar sin contar con todos los datos.
- ✓ Es difícil representar la negación.

**Imperativos vs. Declarativos**

| Imperativos | Declarativos |
|---|---|
| • Se basan en la máquina Von Newman.<br>  • Se preocupa por el ¿Cómo?<br>• No son referencialmente transparentes, es decir, la misma expresión no siempre da los mismos resultados.<br>• El control depende exclusivamente del programador.<br>• Se basan en el lenguaje máquina en el que se apoyan, teniendo como instrucción principal la asignación.<br>• El concepto de variable es un objeto cuyo valor puede cambiar en el tiempo. | • Se basan en modelos matemáticos.<br>  • Se preocupa por el ¿Qué?<br>• Intentan ser referencialmente transparentes, es decir, la misma expresión siempre da los mismos resultados.<br>• El control lo lleva la máquina<br>• Son independientes de la máquina.<br>• Las variables son objetos cuyo valor no se conoce, y que, una vez que se le asocia un valor, conserva dicho valor hasta el final. |

*Comparación de las principales características de cada uno de los enfoques de la Programación.*

**Aplicaciones de la Programación Declarativa**

- ✓ Enrutado de aviones para Alitalia
- ✓ Configuración de teléfonos móviles - Nokia
- ✓ Análisis de ADN y secuencias de proteínas - ICOT / Japón
- ✓ Acceso a Bases de Datos en español Software AG España
- ✓ Configuración de centrales telefónicas - Ericsson
- ✓ Sistema de control de la contaminación del aire - Hungría

**Programación Funcional. Lenguajes Funcionales**

En Programación Funcional —que responde al llamado enfoque denotacional— un programa no es más que una función, obtenida combinando funciones más simples (que a su vez serán combinaciones de funciones aún más simples). Las funciones más simples, que no se pueden descomponer, se denominan primitivas. Un programa será pues, algo de la forma:

$$s = f(e)$$

donde *e* es la entrada y *s* la salida en un momento dado. La ejecución del programa consistirá en evaluar la función *f* para los datos de entrada. Nótese que una función se evalúa, desde un punto de vista formal, sin necesidad de hacer uso de los conceptos de tiempo ni secuencialidad.

**Programación Lógica. Lenguajes Lógicos**

Un programa en Programación Lógica —enfoque axiomático— es un conjunto de proposiciones lógicas (que se asumen como ciertas) expresadas mediante alguna notación. La ejecución en este caso consistirá en comprobar la veracidad (o falsedad) de un aserto, combinando de algún modo las proposiciones dadas inicialmente.

De esta manera surgen dos posibilidades:

- ¿La salida *s* está asociada a la entrada *e*? (Nótese que en este caso una entrada puede ir asociada a múltiples salidas).
- ¿Cuáles son los valores de *s* relacionados con un valor de *e*?

Para poder modelizar matemáticamente estas relaciones se define la función:

$$R : E x S \rightarrow \{0 ,1\}$$

donde *E* es el dominio de las entradas y *S* el dominio de las salidas. Este tipo de función matemática se llama precisamente relación, puesto que enlaza parejas de elementos de ambos conjuntos. Diremos que $a \in E$ está relacionado con $b \in S$ si y sólo si *R(a, b) = 1* (también se suele escribir *aRb*).

**Programación Lógica y Funcional. Dos enfoques no tan diferentes**

Vamos a comentar ahora un par de resultados interesantes para relacionar los dos tipos de programación declarativa:

1. La programación lógica se basa en el empleo y combinación de predicados. Recordemos que un predicado no es más que una función, definida en el conjunto {0, 1}. Por lo tanto, podemos considerar la programación lógica como un subconjunto de la programación funcional.

2. Por otra parte, una función simple

$$f : E \rightarrow S$$

tal que *s = f(e)*, se puede entender como un caso particular de la relación

$$R_f : E\,x\,S \rightarrow \{0\ ,1\}$$

tal que $R_f(e, s) = 1$ <==> *s = f(e)*. Como consecuencia de esto podemos decir que la programación funcional es un subconjunto de la programación lógica.

A la vista de lo anterior podemos afirmar que los enfoques de programación declarativos (lógico y funcional) son equivalentes.