MÓDULO 1 | PROGRAMACION COBOL– Estructuras Básicas

1. INTRODUCCION

El lenguaje **COBOL** (**CO**mmon **B**usiness-**O**riented **L**anguage) (Lenguaje Común Orientado a Negocios) fue inventado entre 1959 y 1961 en Estados Unidos por un equipo liderado por la *Dra Grace Murray Hopper*, doctora en Matemáticas de la Universidad de Yale, con el objetivo de crear un lenguaje de programación universal y user-friendly, que pudiera ser usado fácilmente en cualquier ordenador en el mundo de los negocios, es decir, en *la Informática de Gestión.*

COBOL es un lenguaje de programación que enfatiza la **manipulación de grandes archivos** de datos en una forma relativamente simple, en lugar de preocuparse de procesamientos lógicos o algebraicos extensivos. En contraste con otros lenguajes de programación, Cobol no fue concebido para cálculos complejos matemáticos o científicos (de hecho solo dispone de comandos para realizar los cálculos más elementales), aunque si posee un **elevado grado de precisión y velocidad del cálculo numérico**, pudiendo manejar hasta 30 posiciones decimales. Su empleo es más **apropiado para el proceso de datos en aplicaciones comerciales y la utilización de grandes cantidades de datos.** Y aunque no sea el único lenguaje orientado a éste propósito, si ha sido el mas utilizado en toda la historia.

Al principio, los mainframes eran los tipos de ordenadores escogidos por las grandes empresas, sobre todo las del sector financiero, con sus propios sistemas operativos y sus propios compiladores de Cobol. Estos fueron afianzando al Cobol como un lenguaje perfecto para conseguir sus propósitos por su **robustez, su fiabilidad y su perfecta adaptación a las necesidades de gestión**. En Argentina Cobol es el lenguaje mayormente utilizado en el sector financiero, empresas de servicios públicos (aguas, gas, luz, telefonía), aviación comercial y fuerzas armadas.

Cuando los sistemas operativos empezaron a independizarse de las máquinas, fue cuando los fabricantes de compiladores Cobol comenzaron su expansión.

Así empresas como Liant, Acucorp, Merant, Fujitsu, Nigsun, IBM o Computer Associates, unas más antiguas otras más modernas, nos permiten a los programadores hoy en día seguir programando con éste lenguaje, y que nos sintamos ilusionados con el futuro.

Cobol, es un lenguaje independiente de la plataforma en la que se ejecute, por lo tanto es posible ejecutar el mismo programa sin modificar nada en cientos de sistemas diferentes (Windows, Unix, MS-Dos, Linux, OS400, S36, S34, VMS, Netware, Solaris, etc...). Es un lenguaje que puede comunicarse a la perfección con cualquier base de datos existente en el mercado, así como generar aplicaciones 100%100 Windows. Se puede adaptar a la tecnología cliente-servidor, tecnología de eventos y puede estar presente en la web. En definitiva, se trata de un lenguaje capaz de todo.

EL PRESENTE

"No sé qué lenguajes habrá en el futuro, pero seguro que Cobol estará todavía allí" . Bill Gates.

Siempre hemos estado acostumbrados a asociar los programas en Cobol a los terminales de texto, sin ninguna capacidad gráfica. Eran programas serios, impersonales y aburridos, que daban una impresión muy pobre a los usuarios que los utilizaban. Todo esto ha cambiado, y hoy nos sorprenderíamos al descubrir que muchos de los programas que utilizamos en Windows están realizados en Cobol, y nadie sería capaz de notar ni la más mínima diferencia, simplemente porque no existe.

EL LENGUAJE

Cobol es un lenguaje compilado. Es decir, consta de un código fuente perfectamente legible y adaptado a unas normas, que se puede realizar con cualquier editor de textos y un código objeto (compilado) dispuesto para su ejecución con su correspondiente runtime.

Cuando se ve un programa escrito en Cobol saltan a la vista varios aspectos:

- Existen unos márgenes establecidos que facilitan su comprensión.
- Está estructurado en cuatro partes, cada una de ella con un objetivo dentro del programa.
- La gramática y su vocabulario tienen su base en la lengua inglesa

Hay muchas formas de codificar un programa que cumpla con las especificaciones de diseño. Un programa puede ser mas corto, o correr de manera más eficiente que otro, pero el costo de **mantener un programa** es tan importante para medir su **performance** como lo anterior.

La **programación estructurada** hace que un programa sea más fácil de mantener, ya que los programas hechos según este principio son fáciles de leer, debuggear y mantener, porque siguen ciertas reglas.
Un programa COBOL estructurado está diseñado y escrito en una **jerarquía lógica**, y tiene características como ser:

- **Modularidad** (programas, subprogramas, secciones y párrafos)
- **Identación** consistente que muestra el flujo de control
- **Nombres** de variables y párrafos que describen su función
- **Comentarios** que describan funciones, no implementación
- Líneas en blanco para separar construcciones lógicas.

Un programa estructurado puede ser leído en secuencia, desde el principio hasta el final, ya que no utiliza goto, alter, etc.

El primer paso del diseño de un programa es entender las necesidades que debe cubrir, desarrollando todos los requerimientos y especificaciones que establecen los objetivos del programa.

Las especificaciones para un programa deben describir en detalle las tres partes principales:

Entrada: qué entra al programa
Proceso: qué cambios toman lugar al procesar estos datos
Salida: a qué resultado se debe llegar.

Si te interesa conocer algo más de los comienzos del COBOL puedes ir a:
https://hipertextual.com/2011/12/historia-de-la-tecnologia-el-lenguaje-cobol

2. ESTRUCTURA DEL LENGUAJE COBOL

1.1 ESTRUCTURA DEL PROGRAMA FUENTE

La estructura de los programas fuente se compone de **líneas** de un máximo de 80 caracteres.

Cada una de estas líneas está dividida en **columnas**, que son interpretadas por el compilador de distintas maneras:

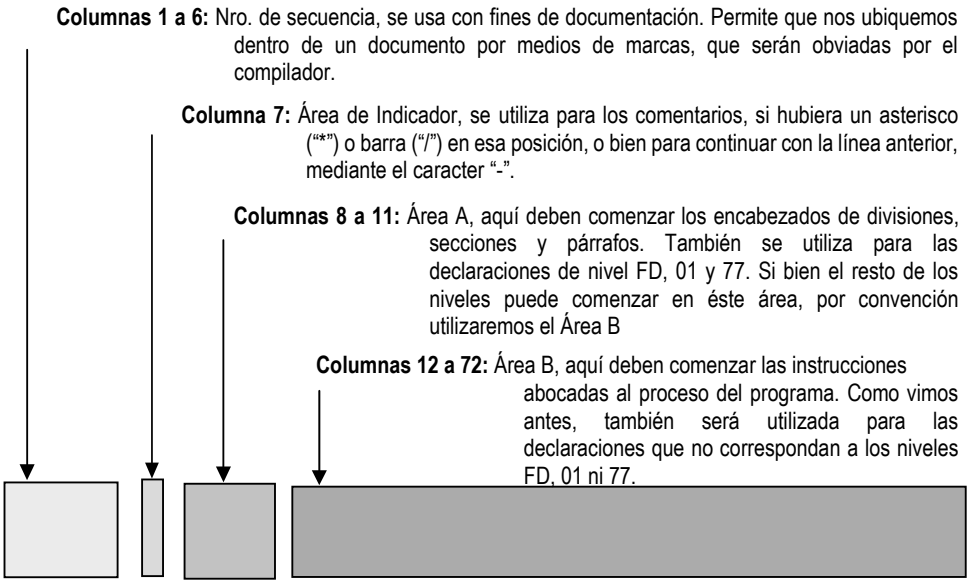


Figura 1 *– Estructura de una línea del programa fuente*

A continuación un ejemplo de la estructura de un programa fuente en Cobol:

```
       1         2         3         4         5         6         7
12345678901234567890123456789012345678901234567890123456789012345678901234567890123...

000001 PROCEDURE DIVISION.
000002*ESTE ES UN EJEMPLO DE LA ESTRUCTURA
000003*DE UN PROGRAMA COBOL
000004 000000-CONTROL.
000005     DISPLAY 'HOLA ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! ! !  COMO
000006-              'ESTAS?'
000007     STOP RUN.
```

Ejemplo 1 *– Estructura del programa fuente*

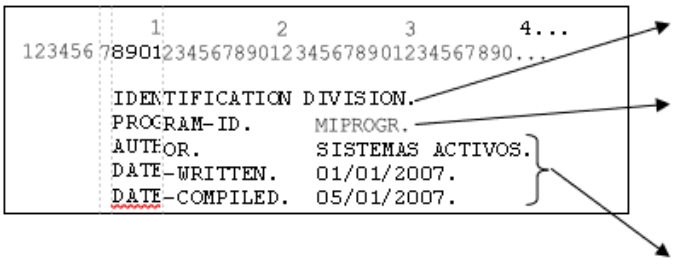
Los programas Cobol se separan en **cuatro divisiones**, cada una tiene una función lógica específica para la resolución del problema.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
PROCEDURE DIVISION.
```

1.1.1 Identification Division

Debe ser la **primer** división en un programa fuente Cobol. Es, además, la única **obligatoria**.

Se utiliza para **especificar el nombre** del programa, y puede incluir la fecha en que fue creado, compilado, y demás **información de documentación**.



Encabezado: También puede declararse como "**ID DIVISION.**", siempre comenzando en el Área A y finalizando con un punto.

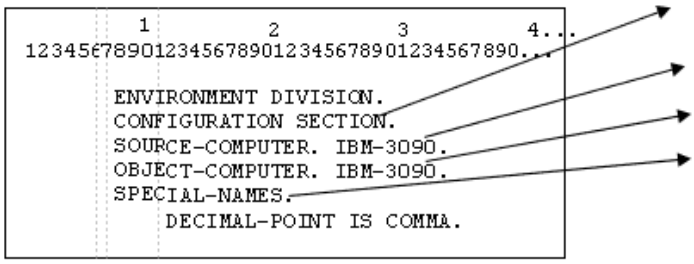
Es el único párrafo[2] **obligatorio** en esta división, debe ser el primero. Aquí se indica el nombre del programa. El párrafo se declara en el Área A, mientras que la información se especifica en el Área B.

Son párrafos opcionales, y no deben respetar ningún orden predeterminado. Indican el autor del programa, fecha de creación y de compilación, respectivamente.

Ejemplo 2 – División de Identificación

1.1.2 Environment Division

En esta división se describe la **configuración** de hardware de la computadora en la cual se compila, y la computadora en la cual se corre el programa objeto. Además, describe la **relación** entre los **archivos** y el **medio** de entrada/salida.



Esta **sección**[1] es **opcional**. Algunos de los **párrafos** [2] que la componen son:

- **Source-Computer:** La computadora en la que el programa es compilado.
- **Object-Computer:** La computadora en la que el programa es ejecutado.
- **Special-Names:** Relaciona palabras definidas por el usuario en el nombres usados por el compilador con programa fuente. Se indica mediante el uso de **cláusulas**. Por ejemplo: **DECIMAL-POINT**, que permite indicar el carácter a utilizar como separador decimal.

Ejemplo 3 – Environment Division – Configuration Section

[1] ***Sección:*** *Es un conjunto de párrafos.*
[2] ***Párrafo:*** *Son divisiones que se utilizan a menudo para definir procedimientos dentro de un programa. Dentro de la PROCEDURE DIVISION, agrupa una cierta cantidad de instrucciones que pueden ser invocadas en forma repetida desde distintos puntos del programa, con sólo indicar que se ejecute el párrafo deseado. El nombre del mismo debe informarse a partir de la columna 8, y finalizar su declaración con un punto.*

Input-Output Section: Esta sección nombra todos los archivos y medios externos que el programa necesita, y provee la información necesaria para la transmisión y manejo de los datos durante la ejecución del programa. Se divide en dos párrafos:

- **File-Control**: se utiliza para nombrar a los archivos que se utilizarán en el programa, relacionándolos con el nombre externo del mismo. Además, brinda la información concerniente al tipo de acceso y organización de los archivos.
  La forma en que se definen los archivos será vista en detalle más adelante.

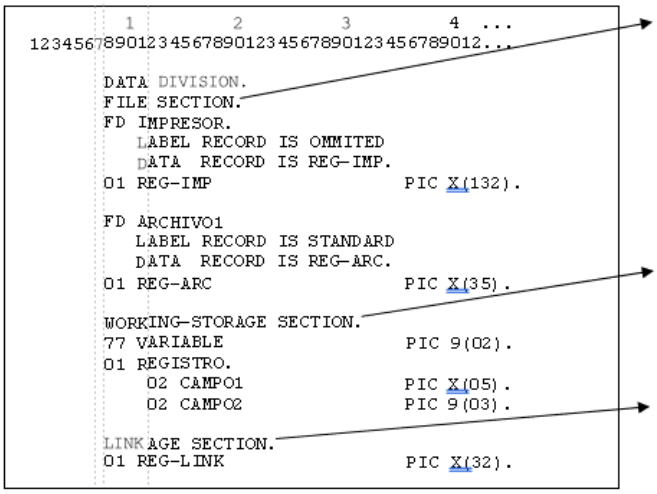
- **I-O-Control**: este párrafo se utiliza para definir los puntos en que se establecerá una re-ejecución del programa, el área de memoria que se compartirá por diferentes archivos, etc. Este párrafo generalmente no es usado.

```
         1         2         3         4         5         6         7
123456 8901234567890123456789012345678901234567890123456789012345678901234567890123...

       ENVIRONMENT DIVISION.
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
           SELECT ARCHIVO1   ASSIGN TO DISK `C:\ARCHIVO.DAT`
                             ORGANIZATION IS LINE SEQUENTIAL.
           SELECT IMPRESOR   ASSIGN TO PRINTER.
```

Ejemplo 4 – Environment Division – Input-Output Section

1.1.3 Data Division



File Section: Aquí se definen las estructuras de datos que se utilizarán para interpretar los registros de los archivos definidos en la Input-Output Section de la Environment Division.
También se pueden especificar parámetros de las impresoras.

Working - Storage Section: Aquí se describen todas las áreas de memoria que no son estructuras de archivos de datos, pero son procesados por el programa.

Linkage Secion: Se utiliza para definir los datos que se utilizan para comunicarse con otros programas, y que son utilizados en ambos

Ejemplo 5 – Data Division

TIPOS DE DATOS

La ***Working-Storage Section*** es utilizada para reservar memoria bajo algún nombre definido por nosotros, que denominaremos etiqueta o identificador. Este proceso es ni más ni menos que la definición de las variables que se utilizarán en el proceso del programa.

En la fracción de memoria asociada a un identificador podremos almacenar información de distinto tipo: Números, letras, datos alfanuméricos (números, letras y símbolos), etc.

Con el fin de especificar el tipo de dato a almacenar en la memoria, se utiliza la cláusula **PICTURE IS**, que se abrevia como **PIC**, y a continuación se especifica el formato de la variable mediante un caracter. En la **Figura 2** se detallan los caracteres aceptados por Cobol para definir los tipos de datos de las variables, y su significado:

Carácter	Tipo representado
9	Digito numérico de (0-9)
A	Letra del alfabeto (A-Z ó a-z)
X	Letra, dígito numérico, o símbolo
S	Signo
V	Separador decimal

Figura 2 – Caracteres para definir el formato de las variables

A continuación se presenta una lista de ejemplos acerca de cómo utilizar la cláusula **PIC** para almacenar distintos tipos de dato:

Rango	Cláusula PIC
0..9	PIC 9
0..999	PIC 999 ó PIC 9(3)
-999..999	PIC S999 ó PIC S9(3)
0,00..9,99	PIC 9V99 ó PIC 9V9(2)
-999,9..999,9	PIC S999V9 ó S9(03)V9
A..Z ó a..z	PIC A
Palabra de 4 letras	PIC AAAA ó PIC A(4)
Texto con puntuaciones de 255 caracteres	PIC X(255)

Ejemplo 6 – Definición de variables en función de los valores a almacenar

DEFINICION DE VARIABLES

Para abordar esta parte de la explicación, se debe tener en cuenta la siguiente premisa: En Cobol **todas** las variables definidas son **globales**.

Para definirlas se utilizan **niveles**, estableciendo una jerarquía, que permite **agruparlas** bajo un mismo nombre (**grupo**), de la misma manera que en Pascal se declaran registros (*record*).

Las variables que no pertenecen a un grupo, es decir, las variables **independientes**, se definen con niveles **77**

Los registros o **grupos** se definen a partir de niveles **01**. Todos estos niveles deben comenzar en el **área A**.

Las variables **dependientes** de otras, deberán tener un nivel mayor al de su contenedor, esto es, niveles desde el **02 al 49.** Como vimos anteriormente, pueden definirse tanto en el área A como en el área B, tomaremos como regla utilizar una identación de 3 espacios entre la variable grupo y las dependientes de la misma.

Para la definición de variables **booleanas** (aquellas que representan solo valores de verdadero o falso), se utiliza el nivel **88.**

Definición de variables independientes

En este ejemplo, definimos tres variables, llamadas **`WSC-COD-CAH, WSC-COD-CTE y WSV-VAR-ALF.`**

```
      1         2         3         4         5         6         7
1234567890123456789012345678901234567890123456789012345678901234567890123...

       77 WSC-COD-CAJ-AHO    PIC S9(02) VALUE 10.
       77 WSC-COD-CTA-CTE    PIC S9(02) VALUE 2.
       77 WSV-VAR-ALF        PIC  X(03) VALUE 'ABC'.
```

Ejemplo 7 – Definición de Variables Independientes

Las dos primeras hacen referencia a los códigos correspondientes a Cajas de Ahorro y Cuentas Corrientes, respectivamente. La tercera es una variable alfabética.

Las variables declaradas como numéricas, en este caso, soportan signo (indicado por el caracter '**S**'), y ocupan dos bytes cada una. Esto quiere decir que pueden almacenar valores comprendidos entre **-99** y **99**. Como vimos anteriormente, también podrían haber sido declaradas como **PIC S99**.

La tercera variable es alfanumérica, indicado mediante el carácter "**X**", y su longitud es de 3 bytes. También podría haberse definido como **PIC XXX**.

La cláusula **VALUE** se utiliza para especificar cuál será el valor inicial de las variables.

Definición de constantes

Cobol no presenta una alternativa para definir constantes, por lo tanto se definen de la misma forma que las variables, teniendo la consideración de no modificar su valor en ninguna línea del código.

Para hacer más claro el uso de constantes, aplicaremos el siguiente estándar:

- Asignar valor mediante la cláusula **VALUE.**
- Su nombre debe comenzar con "WSC-" y hacer referencia a su función, no a su valor.
- No declarar constantes con valor "ceros" o "espacios".

De esta manera, si quisiéramos utilizar como constantes las variables numéricas del ejemplo anterior podríamos hacerlo siempre y cuando no cambiemos su valor. Luego, cuando hagamos referencia al Código de cajas de ahorro no especificaremos el valor "10", sino que utilizaremos el nombre de la constante: WSC-COD-CAJ-AHO.

Además de brindar más claridad al código, el uso de constantes permite que las modificaciones de valores que se utilizan en forma repetida sean más fáciles. Por ejemplo, si el día de mañana el código de cajas de ahorro cambiara a 32, con sólo modificar el valor de la constante WSC-COD-CAJ-AHO lo resolveríamos. Si no utilizáramos constantes, habría que buscar en el código fuente todas las referencias a "Código de caja de ahorro" representadas por el valor 10, y cambiarlas por el valor 32, corriendo el riesgo de omitir alguna o modificar algo que hacía referencia a otra variable.

Definición de Grupos

Otro tema a tratar es el de los registros o grupos. Imaginemos el caso en que manejamos los datos correspondientes a código de cliente y su apellido y nombre.

Tendríamos la siguiente información:
- Código de cliente: Alfabético de 4 posiciones
- Apellido y nombre: Alfabético de 50 posiciones

La definición para estos datos, sería de la siguiente manera:

```
          1         2         3         4         5         6         7
7  123456 8901234567890123456789012345678901234567890123456789012345678901234567890123...

          77 WSV-COD-CLI     PIC X(04) VALUE SPACES.
          77 WSV-APE-NOM     PIC X(50) VALUE SPACES.
```

Ejemplo 8 – Definición de Variables

Sin embargo, ambos datos se refieren a información de un cliente, de modo que seria más práctico definirlos de una manera que represente esta realidad. Se haría de la siguiente manera:

```
          1         2         3         4         5         6         7

8 1234567 9012345678901234567890123456789012345678901234567890123456789012345678901234567890123...

          01 WSV-DAT-CLI.
             05 WSV-COD-CLI     PIC X(04) VALUE SPACES.
             05 WSV-APE-NOM     PIC X(50) VALUE SPACES.
```

Ejemplo 9 – Definición de Variables - Grupos

De esta forma, queda claro que los datos del cliente incluyen su código, y su apellido y nombre. También permite manejar ambos datos al mismo tiempo.

Para entender mejor cómo se trabaja con variables en grupo lo representamos gráficamente:

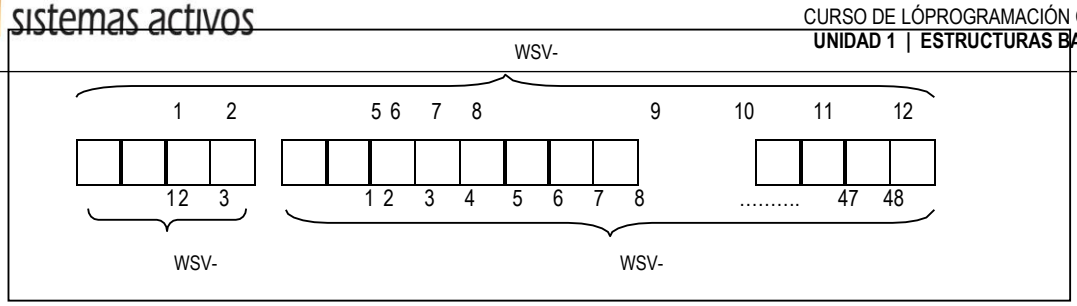


Figura 4– Almacenamiento de las variables en memoria

Como muestra la figura, WSV-DAT-CLI apunta al mismo área de memoria que utilizan WSV-COD-CLI y WSV-NOM-APE. En la declaración no se informa la longitud del nivel superior, dado que equivale a la sumatoria de las longitudes de los campos que componen el grupo.

Si quisiéramos "limpiar" los datos que contiene una variable, deberíamos mover *spaces* a cada uno de los campos. Sin embargo, la estructura de grupo permite simplificar el proceso, moviendo *spaces* al nivel superior que engloba al resto, en este ejemplo, WSV-DAT-CLI.

De acuerdo a lo visto anteriormente, supongamos ahora que el apellido y el nombre del cliente vienen informados en dos variables alfabéticas de 25 posiciones cada una. La definición correspondiente para esta nueva estructura sería:

```
      1         2         3         4         5         6         7
7 123456 8901234567890123456789012345678901234567890123456789012345678901234567890123...

         01 WSV-DAT-CLI.
            05 WSV-COD-CLI     PIC X(04).
            05 WSV-APE-NOM.
               10 WSV-APE-CLI  PIC X(25).
               10 WSV-NOM-CLI  PIC X(25).
```

Ejemplo 10 – Definición de Variables - Grupos

Definición de variables booleanas

Para comprender mejor el funcionamiento de éste tipo de variables, apelaremos al siguiente ejemplo:

El campo WSS-NUMERO es un numérico de 1 byte, por tanto almacenará datos comprendidos entre 0 y 9. Dependiendo de la variable WSS-NUMERO declararemos dos niveles 88 (que corresponde a las variables booleanas): WSS-NRO-PAR, con los valores 2, 4, 6 y 8; y WSS-NRO-IMPAR, con los valores 1, 3, 5, 7 y 9.

WSS-NUMERO puede utilizarse como cualquier otra variable dentro del programa, su particularidad es que dependiendo del valor que se almacene variarán los valores de verdad de sus niveles 88. Es decir, si WSS-NUMERO almacenara 2, 4, 6 u 8, según este ejemplo, la variable WSS-NRO-PAR tomará automáticamente el valor VERDADERO, mientras WSS-NRO-IMPAR valdrá FALSO. Si almacenara un 1, 3, 5, 7 ó 9, se invertirían los valores de verdad de los niveles 88. En el caso de almacenar 0, ambos niveles 88 tendrán valor FALSO dado que ninguno de los dos contempla ése valor en la cláusula VALUE.

Expresada en código, la definición de la variable sería la siguiente:

```
      1         2         3         4         5         6         7
7 123456 8901234567890123456789012345678901234567890123456789012345678901234567890123...

         01 WSS-NUMERO        PIC 9(01) VALUE 0.
            88 WSS-NRO-PAR              VALUE 2 4 6 8.
            88 WSS-NRO-IMPAR            VALUE 1 3 5 7 9.
```

Ejemplo 11 - Definición de Variables Booleanas

Dentro de la lógica del programa, una sentencia de la forma :

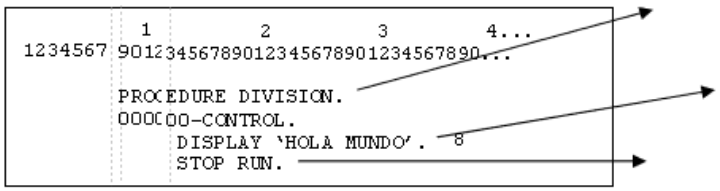
```
IF  WSS-NRO—PAR
    DISPLAY   'ES   NRO
    PAR'
END-IF.
```

Equivale a una sentencia de la forma:

```
IF (WSS-NUMERO = 2 OR WSS-NUMERO = 4 OR WSS-NUMERO = 6 OR WSS-NUMERO = 8) DISPLAY
  'ES NRO PAR'
END—IF.
```

1.1.4 Procedure Division

Aquí es donde se define la **lógica** del programa, mediante **sentencias** que pueden agruparse, como se explicó previamente, en secciones y/o párrafos.



Encabezado: Siempre debe comenzar en el Área A y finalizar con un punto

Sentencia Imperativa: Indica una acción a ser realizada durante la ejecución del programa, en este caso mostrar la leyenda "Hola Mundo".

Sentencia de terminación del programa

Ejemplo 12 – División de Procedimientos

TERMINACIÓN DE UNPROGRAMA

En COBOL existen una serie de instrucciones con las cuales se puede terminar un proceso. Tanto estas instrucciones como las que presentaremos más adelante deben ser codificadas en el AREA B.

STOP RUN

Detiene el proceso del programa y devuelve el control al sistema operativo.

GOBACK

Esta instrucción cede el control al programa que dio origen a la ejecución del programa actual.

SENTENCIAS

Existen, como en todo lenguaje de programación, tres tipos de sentencias:

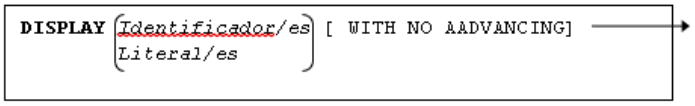
- **Sentencias imperativas**: Indican una acción a desarrollar en el programa.
- **Sentencias Condicionales**: Dan al programa la capacidad de tomar decisiones, permiten caminos alternativos dentro de la ejecución de un proceso.
- **Sentencias de Repetición**: Se utilizan para ejecutar un conjunto de sentencias 1 ó más veces.

Sentencias Imperativas

Explicaremos a continuación un conjunto de instrucciones básicas con las cuales podremos comenzar a trabajar con los primeros programas.

DISPLAY

Escribe los datos que se le indiquen como operandos en el dispositivo de salida, normalmente sucede que es la pantalla, aunque esto no es necesariamente así. La estructura de la sentencia es la siguiente:



Por defecto la sentencia avanza una línea luego de escribir la información. Esto puede evitarse utilizando WITH NO ADVANCING.

Figura 5 – DISPLAY – Formato Básico

ACCEPT

Lee desde el dispositivo de entrada, que normalmente es el teclado, y almacena el dato en la variable que se indique. La estructura de la sentencia es la siguiente:

Formato básico

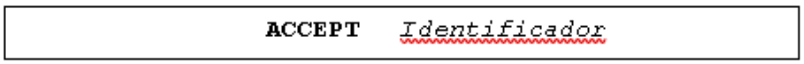


Figura 6 – ACCEPT – Formato Básico

Formato #2

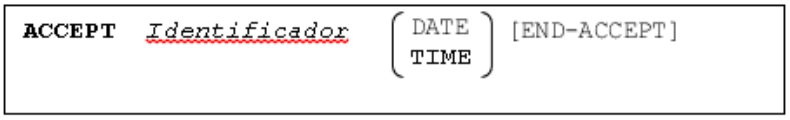


Figura 7 – ACCEPT – Formato 2

Utilizando este formato el ACCEPT obtiene la fecha del sistema, o parte de ella, y la almacena en la variable identificador.

Si utilizamos la palabra reservada DATE obtendremos seis cifras que representan la fecha en formato AAMMDD (AA: Año; MM: Mes; DD: Día).

Si utilizamos TIME obtendremos ocho cifras que representan la hora en formato HHMMSSCC (HH: Horas; MM: Minutos; SS: Segundos; CC: Centésimos).

MOVE

Se utiliza para transferir datos de un identificador a otro. Formato básico:

Luego de ejecutarse la instrucción, Identificador-2 tendrá el mismo valor que Identificador-1, o Literal.

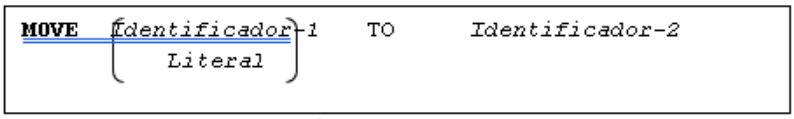


Figura 8 – MOVE

COMPUTE

Esta sentencia permite hacer una asignación de valor ó resolver una expresión aritmética. Formato básico

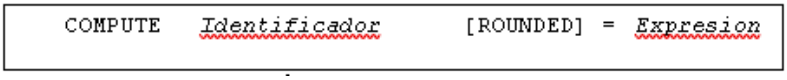


Figura 9 – COMPUTE

La expresión aritmética puede presentar los siguientes operadores:

Operador	Descripción
+	Suma
-	Resta
*	Multiplicación

Operador	Descripción
/	División
**	Potencia
( )	Asociación

Presentaremos a continuación el ejemplo de un programa Cobol que solicita el ingreso por teclado de dos números, los suma, y muestra el resultado obtenido:

```
      1         2         3         4         5         6         7
123456 8901234567890123456789012345678901234567890123456789012345678901234...

       IDENTIFICATION DIVISION.
       PROGRAM-ID.   MIPROGR.
       AUTHOR.       SISTEMAS ACTIVOS.

       DATA DIVISION.
       WORKING-STORAGE SECTION.

       77 NUM1     PIC 9(2).
       77 NUM2     PIC 9(2).
       77 TOTAL    PIC 9(3).

       PROCEDURE DIVISION.
       PROCESO.
           DISPLAY "INGRESE EL PRIMER NÚMERO:"
           ACCEPT NUM1
           DISPLAY "INGRESE EL SEGUNDO NÚMERO:"
           ACCEPT NUM2
           COMPUTE TOTAL = NUM1 + NUM2
           DISPLAY "EL TOTAL ES: " TOTAL
           STOP RUN.
```

Ejemplo 13 – Programa

***Sentencias Imperativas* (Control de Flujo)**

Son las cláusulas que permiten que los programas tomen decisiones, repitan una tarea y analicen condiciones para actuar de forma diferente ante sucesos distintos y/o iterar sobre pasos particulares del proceso mientras no se cumpla determinada condición.

Organización en párrafos

Los párrafos son divisiones que se utilizan para definir procedimientos dentro de un programa, agrupan una cierta cantidad de instrucciones que pueden ser invocadas en forma repetida desde distintos puntos del programa con sólo indicar que se ejecute el párrafo deseado. Al utilizarlos en forma ordenada se puede evitar la redundancia en el código.

El nombre del párrafo debe informarse a partir de la columna 8, y finalizar su declaración con un punto.

PERFORM

La cláusula PERFORM puede utilizarse para invocar párrafos de forma tal que se ejecuten las sentencias que contenga, y al terminar la ejecución del mismo se devuelva el control del programa a la línea siguiente a la que realizó la llamada.

```
      1         2         3         4         5         6         7
12 4567890123456789012345678901234567890123456789012345678901234567890123...
      PROCEDURE DIVISION.
       000000-CONTROL.
           PERFORM 100000-INICIO
           PERFORM 200000-PROCESO
           DISPLAY 'FIN'
           PERFORM 300000-FINAL.

      100000-INICIO.
           DISPLAY 'INICIO'.

      200000-PROCESO.
           DISPLAY 'PROCESO'.

      300000-FINAL.
           STOP RUN.
```

Organización en párrafos

El resultado de la ejecución de este ejemplo será la siguiente salida por pantalla:

```
INICIO
PROCESO
FIN
```

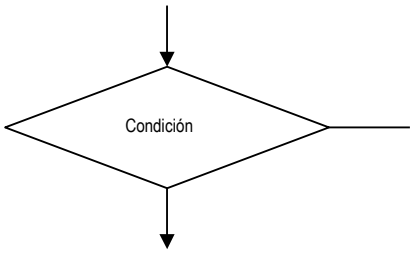
Decisión

IF

Se utiliza para tomar una decisión, para ello se necesita al menos una condición y dos alternativas. Su estructura básica es la siguiente:

```
IF [condición]
   Sentencias-1
[ELSE
   Sentencias-2]
END-IF
```

Para entender mejor su funcionamiento analizaremos el siguiente diagrama donde la cláusula ***IF*** se expresa con un rombo, y dependiendo si se cumple o no la condición indicada dentro del rombo, el flujo toma distintos caminos.



Para componer la condición utilizaremos los siguientes operadores lógicos soportados por COBOL:

Operador	Función
=	Compara si dos expresiones son iguales
NOT =	Compara si dos expresiones son diferentes
>	Compara si un valor es mayor a otro
<	Compara si un valor es menor a otro
>=	Compara si un valor es mayor o igual a otro
<=	Compara si un valor es menor o igual a otro
AND	Multiplicador lógico
OR	Sumador lógico
IS NUMERIC	Verifica si una expresión puede interpretarse como un número

Ejemplo #1

```
      1         2         3         4         5         6         7
123456 8901234567890123456789012345678901234567890123456789012345678901234567890123...

           IF A > 3 AND A < 10
                  DISPLAY "OK"
           ELSE
                  DISPLAY "ERROR"
           END-IF
```

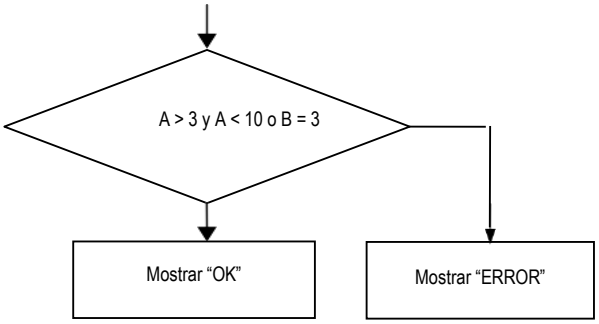
En este ejemplo se evalúa si el valor de la variable A almacena un valor entre 4 y 9. En el caso que el valor de A sea 4, 5, 6, 7, 8 ó 9 veremos “OK” en pantalla, sino “ERROR”. Es importante aclarar que en este ejemplo A debe ser numérica.

Ejemplo #2

```
      1         2         3         4         5         6         7
123456 890.234567890123456789012345678901234567890123456789012345678901234567890123...

           IF A > 3 AND A < 10   OR B = 3
              DISPLAY "OK"
           ELSE
              DISPLAY "ERROR"
           END-IF
```

En este ejemplo se incorporó la condición “si B es igual a 3”. El “OR” indica que si se cumpliera esta condición, o la anterior sin necesidad de que lo hagan las dos al mismo tiempo, obtendremos la palabra “OK” en pantalla. Si no se cumpliera ninguna de las dos condiciones, se mostrará la palabra “ERROR”.
La representación simbólica del flujo de programa para este ejemplo es la siguiente:



EVALUATE

Es una estructura de decisión que puede ser interpretada como un anidamiento de *IF* en forma compacta. Tiene dos formas de uso distintas, la primera permite evaluar una variable, mientras la segunda permite evaluar condiciones. En ambos casos, al igual que con la cláusula *IF*, se toman diferentes acciones dependiendo de la condición que se cumpla.

Primera estructura

```
7        1         2         3         4         5         6         7
 123456 8901234567890123456789012345678901234567890123456789012345678901234567890123...

           IF A = 1
              DISPLAY "1"
           ELSE
              IF A = 2
                 DISPLAY "2"
              ELSE
                 IF A = 3
                    DISPLAY "3"
                 ELSE
                    DISPLAY "X"
                 END-IF
              END-IF
           END-IF
```

Estructura con IF

```
7        1         2         3         4         5         6         7
 123456 8901234567890123456789012345678901234567890123456789012345678901234567890123...

           EVALUATE A
           WHEN 1
                 DISPLAY "1"
           WHEN 2
                 DISPLAY "2"
           WHEN 3
                 DISPLAY "3"
           WHEN OTHER
                 DISPLAY "X"
           END-EVALUATE
```

Estructura con EVALUATE

Segunda estructura

```
                2 3         4         5         6         7
123             234567890123456789012345678901234567890123456789012345678901234567890123...
456
                IF A = 1 AND B = 2
                DISPLAY "1"
                ELSE
                IF A = 2 OR B = 3
                DISPLAY "2"
                ELSE
                IF B = 4
                DISPLAY "3"
                ELSE
                DISPLAY "4"
                END-IF
                END-IF
                END-IF
```

Estructura con IF

```
          1         2         3         4         5         6         7
123456 8901234567890123456789012345678901234567890123456789012345678901234567890123...

           EVALUATE TRUE
           WHEN A = 1 AND B = 2
                   DISPLAY "1"
           WHEN A = 2
           WHEN B = 3
                   DISPLAY "2"
           WHEN B = 4
                   DISPLAY "3"
           WHEN OTHER
                   DISPLAY "4"
           END-EVALUATE
```

Estructura con EVALUATE

Cabe destacar el caso en que se simula el OR en el EVALUATE como una superposición de dos cláusulas WHEN.

Iteración

Consiste en una serie de pasos del programa que son repetidos n veces, dependiendo de una condición. Dependiendo de la variante de la cláusula ***PERFORM*** que se utilice, será la forma de armar la iteración y evaluar la condición.

PERFORM UNTIL-WITH TEST AFTER

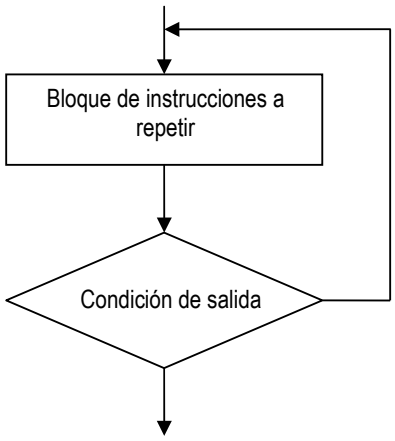
Permite repetir un proceso ***hasta*** que se cumpla una condición específica. En el siguiente ejemplo, un ciclo de repetición muestra en pantalla los números del 1 al 10.

```
                2 3         4         5         6         7
123             234567890123456789012345678901234567890123456789012345678901234567890123...
456
                MOVE 1 TO NUM
                PERFORM WITH TEST AFTER UNTIL NUM > 10
                DISPLAY NUM
                COMPUTE NUM = NUM + 1
                END-PERFORM
```

Estructura de repetición HASTA

Al comenzar el proceso se inicializa la variable NUM en 1. En la siguiente sentencia se establece la condición para que se repita el ciclo hasta que la variable NUM sea mayor a 10. Dentro de la estructura ***PERFORM*** establecemos lo que queremos repetir, en este caso mostrar el valor de NUM y luego incrementar la variable en 1.
En el ejemplo la estructura que se utiliza es ***PERFORM WITH TEST AFTER*** **UNTIL**, lo cual se traduce como "Hacer, evaluando al final, hasta que", es decir que la evaluación de la condición se realizará luego de procesar al menos una vez las instrucciones que componen el ciclo.

Expresado en diagrama, el flujo de programa es el siguiente:



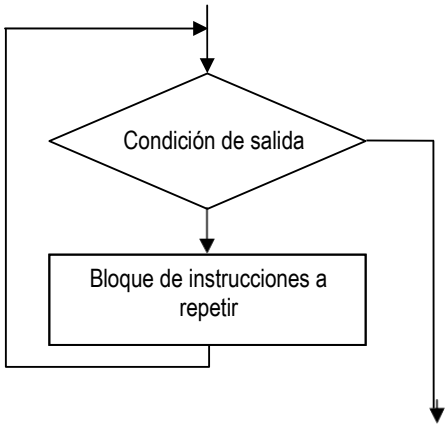
PERFORM UNTIL

Esta forma de la cláusula PERFORM permite repetir un proceso ***mientras*** que no se cumpla una condición específica. Utilizaremos el mismo ejemplo a fin de contrastar las diferencias entre ambas formas.

```
          1         2         3         4         5         6         7
7 123456 8901234567890123456789012345678901234567890123456789012345678901234567890123...

         MOVE 1 TO NUM
         PERFORM UNTIL NUM > 10
                 DISPLAY NUM
                 COMPUTE NUM = NUM + 1
         END-PERFORM
```

Estructura de repetición MIENTRAS



Al desaparecer de la cláusula *WITH TEST AFTER*, se evaluará la condición antes de procesar las sentencias dentro de la estructura del *PERFORM.* A diferencia del ejemplo anterior, si la condición no se cumpliera el ciclo nunca se ejecutaría.

La forma de representar el ejemplo mediante un diagrama es la siguiente:

PERFORM FROM-UNTIL

Esta forma se utiliza cuando se conocen los valores que debe tomar cierta variable. En los casos anteriores pretendíamos que la variable NUM recorra los valores del 1 al 10, se podría haber expresado de la siguiente manera:

```
         1         2         3         4         5         6         7
123456 8901234567890123456789012345678901234567890123456789012345678901234567890123...

           PERFORM VARYING NUM FROM 1 BY 1 UNTIL NUM > 10
                   DISPLAY NUM
           END-PERFORM
```

Estructura de repetición DESDE-HASTA

En el siguiente diagrama de flujo se representa el ejemplo, y se relacionan las sentencias del diagrama con las partes de la cláusula utilizada: