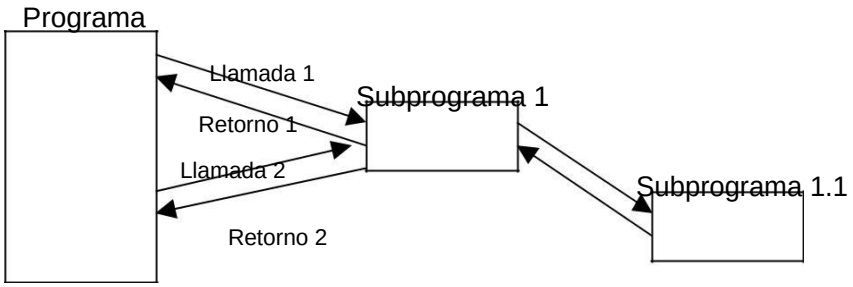
2- Subalgoritmos o Subprogramas

Refrescando conceptos ya vistos en años anteriores, recordemos que un subprograma puede realizar las mismas acciones que un programa: 1) aceptar datos; 2) realizar cálculos; y 3) devolver resultados. Sin embargo un subprograma se utiliza por un programa para un propósito específico. El subprograma recibe datos desde el programa y le devuelve resultados.

Se dice que el programa principal *llama* o *invoca* al subprograma. El subprograma ejecuta una tarea específica y *devuelve* el control al programa. Esto puede suceder en diferentes lugares del programa, y a su vez un subprograma puede llamar a sus propios subprogramas.



Existen, como recordarán, dos tipos de subprogramas: *funciones* y *procedimientos*.

2.1 Funciones

Matemáticamente una función es una operación que toma uno o más valores llamados *argumentos* y produce un valor denominado *resultado* –valor de la función para los argumentos dados-. Todos los lenguajes de programación tienen *funciones incorporadas* o *intrínsecas* y funciones definidas por el mismo usuario, o *funciones externas*.

Cuando las funciones estándares o internas no permiten realizar el tipo de cálculo deseado es necesario recurrir a las funciones externas, que pueden ser definidas por el usuario mediante una *declaración de función*.

A una función no se la llama explícitamente, sino que se la invoca o referencia mediante un nombre y una lista de parámetros actuales.

El algoritmo o programa llama o invoca a la función con su nombre en una expresión seguida de una lista de argumentos que deben coincidir en cantidad, tipo y orden con los de la función que fue definida. La función devuelve un único valor.

Una función puede ser llamada de la siguiente forma:

`Nombre_función` (*lista de parámetros actuales*)

Los argumentos de la declaración de la función se denominan *parámetros formales*, *ficticios* o *mudos*; son nombres de variables de otras funciones o procedimientos y solo se utilizan dentro del cuerpo de la función. Los argumentos usados en la llamada a la función se denominan *parámetros actuales*, que a su vez pueden ser variables, constantes, expresiones, valores de funciones o nombres de otras funciones.

Cada vez que se llama a la función desde el programa principal se establece automáticamente una correspondencia entre los parámetros formales y los parámetros actuales. Debe haber exactamente el mismo número de parámetros actuales que de parámetros formales en la declaración de la función y se presupone una correspondencia uno a uno de izquierda a derecha entre los parámetros formales y los actuales.

Una llamada a la función implica los siguientes pasos:

1) A cada parámetro formal se le asigna el valor real de su correspondiente parámetro actual.
2) Se ejecuta el cuerpo de acción de la función.
3) Se devuelve el valor de la función al nombre de la función y se retorna al punto de llamada.

2.2 Procedimientos

Aunque las funciones son herramientas de programación muy útiles para la resolución de problemas, su alcance está muy limitado. Con frecuencia se requieren subprogramas que calculen varios resultados en vez de uno solo, o que realicen la ordenación de una serie de números, etc. En estas situaciones una *función* no es apropiada y se necesita disponer del otro tipo de subprogramas: los *procedimientos*.

Un procedimiento es un subprograma que ejecuta un proceso específico. Ningún valor está asociado con el nombre del procedimiento; por consiguiente no puede ocurrir en una expresión. Un procedimiento se llama escribiendo su nombre seguido de los parámetros actuales

Los parámetros formales de la declaración del procedimiento tienen el mismo significado que en las funciones; los parámetros variables –en aquellos lenguajes que lo soportan, por ej. Pascal- están precedidos cada uno de ellos por la palabra **`var`** para designar que ellos obtendrán resultados del procedimiento en lugar de los valores actuales asociados a ellos.

La lista de parámetros, bien formales en el procedimiento o actuales en la llamada al mismo, se conocen como lista de parámetros, y estas pueden ser de Entrada, de Salida o de Entrada/Salida.

2.3 Ámbito: Variables Locales y Globales

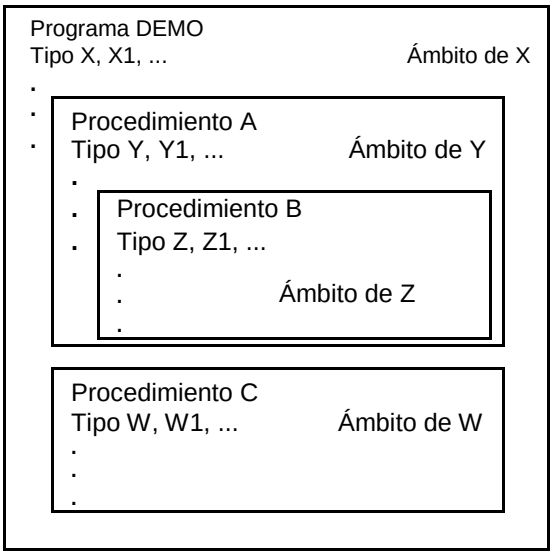
Las variables utilizadas en los programas principales o subprogramas se clasifican en dos tipos: *variables globales* y *variables locales*.

Una *variable local* es aquella que está declarada y definida dentro de un subprograma, en el sentido de que está dentro de ese subprograma y es distinta de las variables con el mismo nombre declaradas en cualquier parte del programa principal. *El significado de una variable local se confina al procedimiento en el que está declarada.* Cuando otro subprograma utiliza el mismo nombre se refiere a otra posición en memoria. Se dice que tales variables son locales al subprograma en el que están declaradas.

Una *variable global* es aquella que está declarada para el programa principal, del que dependen todos los subprogramas.

La parte del programa/algoritmo en que una variable se define se conoce como *ámbito*.

El uso de variables locales tiene muchas ventajas. En particular, hace a los subprogramas independientes, con la comunicación entre el programa principal y los subprogramas manipulados estructuralmente a través de la lista de parámetros. Para utilizar un procedimiento solo necesitamos conocer lo que hace y no tenemos que estar preocupados por su diseño, es decir, cómo están programados.



Ámbito de Identificadores

Una variable local a un subprograma no tiene ningún significado en otro subprograma. Si un subprograma asigna un valor a una de sus variables locales, este valor no es accesible a otros programas, es decir, no pueden utilizar este valor. A veces, también es necesario que una variable tenga el mismo nombre en diferentes subprogramas.

Por el contrario, las variables globales tienen la ventaja de compartir información de diferentes subprogramas sin una correspondiente entrada en la lista de parámetros.

El *ámbito de un identificador* es la parte del programa donde se conoce el identificador. Las variables definidas en un ámbito son accesibles en el mismo, es decir, en todos los procedimientos interiores.

2.4 Comunicación con subprogramas: Paso de parámetros

Cuando un programa llama a un subprograma, la información se comunica a través de la lista de parámetros y se establece una correspondencia automática entre los parámetros formales y actuales. *Los parámetros actuales son sustituidos o utilizados en lugar de los parámetros formales.*

La declaración del subprograma se hace con:

```
Procedimiento nombre (clase tipo_de_dato: F1
                      clase tipo_de_dato: F2
                      ......................
                      clase tipo_de_dato: Fn)
   .
   .
   .
fin_procedimiento
```

Y la llamada al subprograma con:

```
Nombre (A1, A2, ... An);
```

Donde F1, F2, ... Fn son los parámetros formales y A1, A2, ... An los parámetros actuales o reales.

Existen dos métodos para establecer la correspondencia de parámetros:

1. *Correspondencia posicional*. La correspondencia se establece aparejando los parámetros reales y formales según su posición en la lista: así, Fi se corresponde a Ai, donde i=1, 2, ... n. Este método se complica en legibilidad cuando el número de parámetros es muy grande.
2. *Correspondencia por el nombre explícito*. En este método, en la llamada se indica explícitamente la correspondencia entre los parámetros reales y formales. (Ada)
   Por ej.:`Sub(Y→B,X→30)`

Por lo general, la mayoría de los lenguajes usan exclusivamente la correspondencia posicional, y es el que usamos nosotros en la materia.

Los métodos más empleados para realizar el paso de parámetros son: *paso por valor* y *paso por referencia*.

2.4.1 Paso por valor

Este método se utiliza en muchos lenguajes de programación: por ej., C, Modula-2, Pascal, Algol y Snobol. La razón de su popularidad es la analogía con los argumentos de una función, donde los valores se proporcionan en el orden de cálculo de resultados. Los parámetros se tratan como variables locales y los valores iniciales se proporcionan copiando los valores de los correspondientes argumentos.

Cualquier cambio realizado en los valores de los parámetros formales durante la ejecución del subprograma se destruyen cuando se termina el subprograma.

2.4.2 Paso por parámetro

En numerosas ocasiones se requiere que ciertos parámetros sirvan como parámetros de salida, es decir, se devuelvan los resultados a la unidad o programa que llama. Este método se denomina

paso por parámetro. La unidad que llama pasa a la unidad llamada la dirección del parámetro actual (que está en el ámbito de la unidad llamante). Una referencia al correspondiente parámetro formal se trata como una referencia a la posición de memoria, cuya dirección se ha pasado. Entonces una variable pasada como parámetro real es compartida, es decir, se puede modificar directamente por el subprograma.

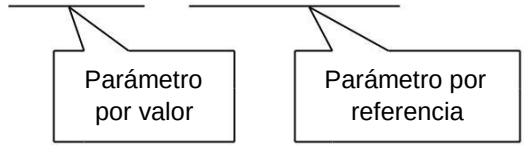
La característica de este método se debe a su simplicidad y su analogía directa con la idea de que las variables tienen una posición de memoria asignada desde la cual se pueden obtener o actualizar sus valores.

En este método los parámetros son de entrada/salida y se denominan parámetros variables.

Los parámetros valor y los parámetros variable se suelen definir en la cabecera del subprograma.

En el siguiente ejemplo, en el mismo subprograma se especifican parámetros por valor y por referencia:

```
Procedimiento Ejemplo (i: entero; var j: entero);
      .
      .
      .
Fin_procedimiento
```



2.4.3 Síntesis de la transmisión de parámetros

Ya dijimos que los métodos más comunes de transmisión de parámetros son *por valor* y *por referencia.*

El paso de un parámetro por valor significa que el valor del argumento –*parámetro actual o real*- se asigna al parámetro formal. En otras palabras, antes de que el subprograma comience a ejecutarse, el argumento se evalúa a un valor específico (por ejemplo 8 o 12). Este valor se copia entonces en el correspondiente parámetro formal dentro del subprograma.

Una vez que el procedimiento arranca, cualquier cambio del valor de tal parámetro formal no se refleja en un cambio en el correspondiente argumento. Esto es, cuando el subprograma se termine, el argumento actual tendrá exactamente el mismo valor que cuando el subprograma comenzó, con independencia de lo que haya sucedido al parámetro formal. Estos parámetros de entrada se denominan *parámetros valor*.

El paso de un *parámetro por referencia* o *dirección* se llama *parámetro variable*. En este caso, la posición o dirección de memoria (no el valor) del argumento o parámetro actual se envía al subprograma. Si a un parámetro formal se le da el atributo de parámetro variable, entonces un cambio en el parámetro formal se refleja en un cambio en el correspondiente parámetro actual, ya que ambos tienen la misma posición de memoria.

3- Recursividad

Un *procedimiento* o *función recursiva* es aquella que se llama a sí misma. Esta característica permite a un procedimiento recursivo repetirse con valores diferentes de parámetros. La recursión es una alternativa a la iteración.

Normalmente, una solución recursiva es menos eficiente en términos de tiempo de ejecución que una solución iterativa, debido al tiempo adicional de llamadas a procedimientos.

Pero en muchos casos, la recursión permite especificar una solución más simple y natural para resolver un problema que en otro caso sería difícil. Por ello, la recursividad es una herramienta muy potente para la resolución de problemas y la programación.

Se dice que un objeto es recursivo si forma parte de sí mismo o se define en función de sí mismo.

La recursión es un método general de resolver los problemas, reduciéndolos a problemas más simples de un tipo similar. La herramienta necesaria y suficiente para expresar los programas recursivamente es el *procedimiento* o *subrutina*, pues permite dar a una proposición un nombre con el cual puede ser llamado.

Si un procedimiento P contiene una referencia explícita a sí mismo, se dice que es *directamente recursivo*; si P tiene una referencia a otro procedimiento Q, que incluye una referencia (directa o indirecta) a P, entonces se dice que P es *indirectamente recursivo*.

Ej.: Sea la función $F(x)= 2\ F(x-1) + x^2$

$F(0)= 0$

$F(1)= 2\ F(0) + 1^2 = 1$

$F(2)= 2\ F(1) + 2^2 = 6$

...

función F (x:entero):entero

Si x=0 entonces
F=0;

} CASO BASE: valor para el cual la función se conoce directamente sin necesidad de la recursión.

sino
F=2*F(x-1)+x^2

} Llamada Recursiva

- Siempre debe estar definida una condición que implique la terminación.
- Siempre se deben tener en cuenta uno o más casos base, que se puedan resolver sin recursión.
- Progreso: para los casos que deben resolverse recursivamente, la llamada recursiva siempre debe tender a un caso base.

Se acostumbra asociar un conjunto de objetos locales a un procedimiento, esto es, un conjunto de variables, constantes, tipos y procedimientos que se definen localmente a este procedimiento y que carecen de existencia o significado fuera del mismo. Cada vez que ese procedimiento se activa de modo recursivo, se crea un nuevo conjunto de variables locales acotadas.

El problema principal con la recursión reside en los costos ocultos de su funcionamiento interno; aunque estos costos casi siempre son justificables, porque los programas recursivos no solo simplifican el diseño del algoritmo sino que también tienden a proporcionar un código más claro. La recursión nunca debe usarse para sustituir un simple ciclo FOR.

En la aplicación práctica es obligatorio demostrar que la profundidad final de la recursión no solo es finita, sino que en realidad es muy pequeña. Ello se debe a que, luego de cada activación recursiva de un procedimiento P, cierta cantidad de memoria se necesita para alojar sus variables (cuidado con el overflow y el underflow).

La mayoría de los ejercicios que se resuelven de manera recursiva, se pueden resolver por medio de estructuras de control iterativas.

Mientras <CONDICIÓN> hacer <ACCIÓN>	Procedimiento P (parámetros) Si <CONDICIÓN> entonces <ACCIÓN> P(parámetros)
Repetir <SENTENCIA> hasta <CONDICIÓN>	Procedimiento P (parámetros) <SENTENCIA> Si no<CONDICIÓN> entonces <ACCIÓN> P(parámetros)
Para y:=1 a n hacer <SENTENCIA>	Procedimiento P (i) Si i<= n entonces <SENTENCIA> P(i+1)

Estas equivalencias no desvirtúan la naturaleza de la recursión, sino que permiten apreciar la posibilidad de prescindir de la iteración para modelar problemas resolubles mediante bucles y repeticiones.

3.1 ¿Cuándo no usar la recursión?

Los algoritmos recursivos son convenientes cuando el problema o los datos que deben manipularse están definidos en términos recursivos. Con todo, ello no significa que tales definiciones recursivas garantice que un algoritmo recursivo es la mejor manera de resolver el problema.

No se debe utilizar la recursión cuando la iteración ofrece una solución obvia. Sin embargo, esto no debe hacer que evitemos sistemáticamente el uso de la recursión.

La recursión se presta a muchas aplicaciones. El hecho de que los procedimientos recursivos se realicen en máquinas esencialmente no recursivas demuestra que, en la práctica, todo programa recursivo puede transformarse en uno totalmente iterativo. Sin embargo para ello se requiere el manejo explícito de una pila de recursión, y esas operaciones a menudo oscurecen la esencia de un programa, al grado que se torna difícil de entender.

Cuando nos encontramos con algoritmos de naturaleza más bien recursiva que iterativa, debemos formular los procedimientos recursivos. En los casos en que las estructuras de datos hacen obvia y natural la elección de soluciones recursivas se debe utilizar esta técnica.

La eficacia de un algoritmo recursivo viene medida por una serie de factores:

- *Tiempo de ejecución*
- *Espacio de memoria ocupado por el algoritmo*
- *Legibilidad y facilidad de comprensión*

Desde el punto de vista del tiempo de ejecución, los algoritmos recursivos son más lentos y ocupan mayor espacio en memoria, con el inconveniente que puede suponer en el caso de poca memoria disponible.

Sin embargo, en ciertos problemas, la recursividad conduce a soluciones que son mucho más fáciles de leer y comprender que su correspondiente solución iterativa; en estos casos la ganancia en claridad puede compensar con creces el coste en tiempo y en memoria de la ejecución de programas recursivos.

Las definiciones recursivas se caracterizan por su elegancia y simplicidad en contraste con la dificultad y falta de claridad de las definiciones repetitivas.

Algunos programadores utilizan la recursividad como una herramienta conceptual, por su simplicidad y fácil comprensión; y una vez que han escrito la forma recursiva, buscan la forma de convertirla en una versión iterativa y así ganar en eficiencia, en tiempo de ejecución y ocupación de memoria.

De todas maneras, numerosos problemas son difíciles de resolver con soluciones iterativas, y sólo la solución recursiva conduce a la resolución del problema (por ejemplo, Torres de Hanoi o recorrido de Árboles).