ESCUELA DE PREPARACIÓN DE OPOSITORES
E. P. O.
C/. La Merced, 8 – Bajo A Telf.: 968 24 85 54
30001 MURCIA

INF25-SAI27

Programación estructurada. Estructuras básicas. Funciones y procedimientos.

Esquema.

1 Introducción.

La programación estructurada se basa en el teorema de la estructura de Bohm y Jacopini, y establece que cualquier programa puede escribirse utilizando solamente las

tres estructuras lógicas básicas de secuencia, selección y repetición. Cuando se sigue una de estas estructuras desde el principio hasta el final, el programa o módulo que se construye, combinando las tres estructuras básicas, tendrá sólo una entrada y una salida y se leerá desde el principio al fin. La programación estructurada especifica la codificación independiente de los módulos utilizando sólo las tres estructuras lógicas fundamentales y además facilita las técnicas para el diseño de la lógica del módulo interno y guías o normas de codificación para el estilo del programa.

2 Programación estructurada.

En las sucesivas fases de confección de un programa sabemos que la depuración y puesta a punto, junto con el mantenimiento continuo, constituyen uno de los problemas más graves con que se enfrenta el programador, la comunicación.

Tras finalizar un programa, se ha de proceder a comprobar su buen funcionamiento. Normalmente se prueba con un conjunto de datos de ensayo y se analizan los resultados. En función de la bondad y exactitud de estos últimos, el programa se da por bueno o se concluye con la evidencia de la existencia de errores. No obstante, los datos de ensayo no demuestran la ausencia de errores.

La validez real de un programa sólo se podría dar haciendo una prueba exhaustiva con todo el rango posible de valores de entrada, cosa realmente imposible en la mayoría de los casos. Por otra parte, como ya se comentó anteriormente, la comunicación usuario/máquina no sólo se manifiesta en la fase de depuración del programa, sino también en la fase de mantenimiento. Un programa normalmente debe ser modificado cada cierto tiempo y en muchas ocasiones las modificaciones han de ser realizadas por terceras personas que no intervinieron en su diseño.

Un mal desarrollo de un programa lleva a la falta de fiabilidad y pérdida de eficiencia considerables, así como a una disminución de la flexibilidad y por consiguiente la documentación de los programas.

Para aumentar la eficiencia de la programación y el mantenimiento, se necesita dotar a los programas de una estructura. Las razones para ello no sólo es el aumento de fiabilidad y eficiencia, sino también asegurar que los programas sean adaptables, manejables, fácilmente comprensibles y portables. A estas condiciones se les suele añadir la claridad y simplicidad.

A medida que la tecnología electrónica ha ido avanzando, las máquinas se han ido haciendo más rápidas, y el postulado que pervivía en la década de los años sesenta e inclusive los setenta en el que el programador debía arañar tiempo de ejecución recurriendo a trucos de programación, ha ido dejando de tener peso y hoy día se buscan programas claros en su estructura, adaptables, portables (posibilidad de pasarlos de una máquina a otra) y simples. La hora de programador se ha elevado en coste considerablemente y si a eso se le une que las posibles modificaciones en un programa pueden producir la pérdida de claridad del mismo, es preciso realizar el programa con técnicas de programación estructurada que permitan aumentar la eficiencia de la programación y su mantenimiento.

Los primitivos propósitos de la programación estructurada dirigían sus esfuerzos a buscar modos de minimizar la probabilidad de error en el proceso de programación. El factor humano es una de las fuentes más importantes en la comisión de errores. Uno de los objetivos de la programación estructurada es la minimización del error humano.

Aunque al profesor Edgar W. Dijkstra se le considera el padre de la programación estructurada, muchos otros investigadores han trabajado en su desarrollo, y de ellos podemos citar a Hoare, Wirth, Knuth, Dahl, Bolirn, Jacopini, Warnier, etc.

Es difícil dar una definición de programación estructurada por no existir generalmente una definición que sea aceptada a todos los niveles. Podríamos por ello enunciar la programación estructurada como una técnica de construcción de programas que utiliza al máximo los recursos del lenguaje, limita el conjunto de estructuras aplicables a leer y presenta una serie de reglas que coordinan adecuadamente el desarrollo de las diferentes fases de la programación.

Aunque, como decíamos antes, la definición anterior es una de las que más hemos visto escritas, son diferentes las definiciones que se podrían hacer; sin embargo, con lo que sí hay un acuerdo casi total es que la programación estructurada utiliza en su diseño los siguientes conceptos o principios fundamentales recogidos esencialmente en la definición anterior:

- Estructuras básicas.
- Recursos abstractos.
- Diseño descendente (*top-down*).

3 Estructuras básicas.

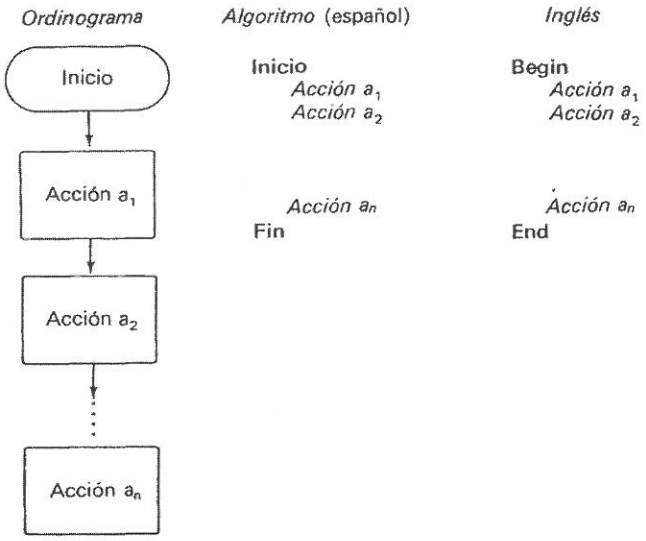
El Teorema de la estructura (o antiguo teorema de Bohm y Jacopini, 1966) demostró que cualquier programa con un solo punto de entrada y un solo punto de salida puede resolverse con tres únicos tipos de estructuras de control: secuencial, alternativa, y repetitiva.

De aquí parte la célebre frase de Dijkstra: "la estructura GO TO es perjudicial para la programación".

3.1 Estructura secuencial.

Una estructura es aquella que ejecuta las acciones sucesivamente unas a continuación de otras sin posibilidad de omitir ninguna y naturalmente sin bifurcaciones. Todas estas estructuras tendrán una entrada y una salida.

Si las acciones son a_1, a_2, ..., a_n, la representación secuencial sería:



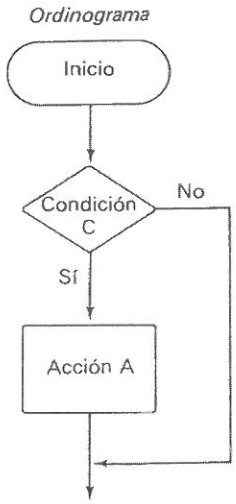
3.2 Estructura alternativa.

Es aquella estructura en la que únicamente se realiza una alternativa (una determinada secuencia de instrucciones) dependiendo del valor de una determinada condición o predicado.

Las estructuras alternativas, también llamadas condicionales, pueden ser de tres tipos: simple, doble, múltiple.

3.2.1 Estructura alternativa simple

Es aquella en que la existencia o cumplimiento de la condición implica la ruptura de la secuencia y la ejecución de una determinada acción.

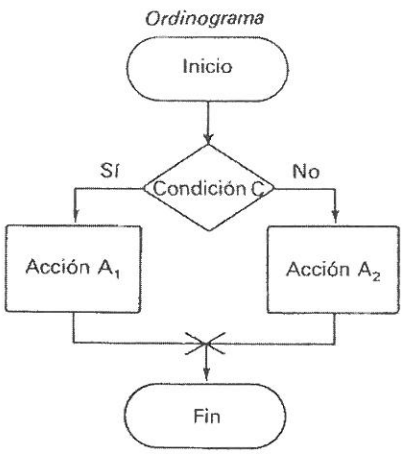


Algoritmo (español)

Si {*condición*}
  Entonces <*acción*}
fin_si
Algoritmo (inglés)
If {*condición*}
  Then {*acción*}
End_if

3.2.2 Estructura alternativa doble.

Es aquella que permite la elección entre dos acciones o tratamientos en función de que se cumpla o no determinada condición.



Algoritmo (español)

Si {*condición*}
  Entonces {*acción* A_1}
Si_no {*acción* A_2}
Fin_si

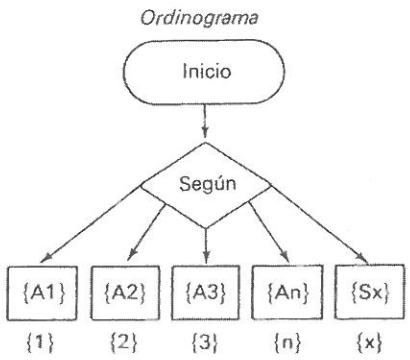
Algoritmo (inglés)
If {*condición*}
  Then {*acción* A_1}
  Else {*acción* A_2}
End_if

3.2.3 Estructura alternativa múltiple.

Las estructuras alternativas múltiples se adoptan cuando la condición puede tomar *n* valores enteros distintos: 1, 2, 3, ..., *n*.

Según se elija uno de estos valores en la condición, se realizará una de las *n* acciones (cada vez sólo se ejecuta una acción). Esta estructura propuesta por HOARE es la `case` del lenguaje Pascal o `switch` del lenguaje C.

Esta estructura puede seleccionar *n* acciones según sea el valor que toma la variable de control.



Algoritmo (español)
Según {variable}
{*condición-1*} : {*acción* A_1}
{*condición-2*} : {*acción* A_2}
{*condición-3*} : {*acción* A_3}
.
.
{*condición-n*} : {*acción* A_n}
Otros : {*acción Sx*}
Fin_según

Algoritmo (inglés)
Case
.
.
End_case

3.3 Estructuras repetitivas.

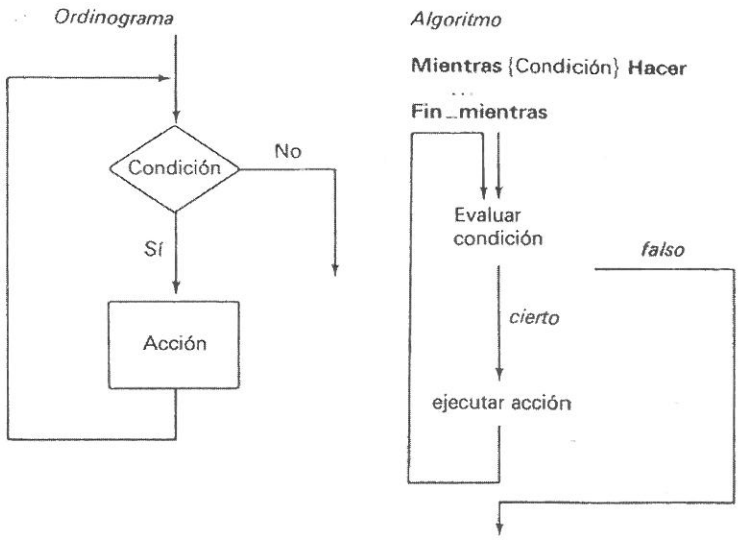
Las estructuras repetitivas o iterativas son aquéllas en las que las acciones se ejecutan un número determinado de veces y dependen de un valor predefinido o el cumplimiento de una determinada condición. Las estructuras repetitivas permiten representar aquellas acciones que pueden descomponerse en otras subacciones primitivas.

Una iteración es el hecho de repetir la ejecución de una secuencia de acciones o de una acción. Un bucle o lazo es el conjunto de acciones iterativas. Para describir una iteración, si se conoce el número *n* de repeticiones, se puede escribir simplemente *n* veces la acción o secuencia de acciones a repetir. Sin embargo, si *n* es grande, las operaciones anteriores pueden resultar tediosas y la secuencia algorítmica difícil de leer. Con frecuencia, es difícil determinar el número de repeticiones. En consecuencia, es preciso disponer de estructuras algorítmicas que permitan describir una iteración de forma cómoda. Las tres estructuras más usuales, dependiendo de que la condición se encuentre al principio o al final de la iteración, son: la estructura mientras, la estructura repetir-hasta, y la estructura para (desde-hasta)

3.3.1 Estructura mientras.

El bucle mientras determina la repetición de un grupo de instrucciones mientras la condición se cumpla inicialmente.

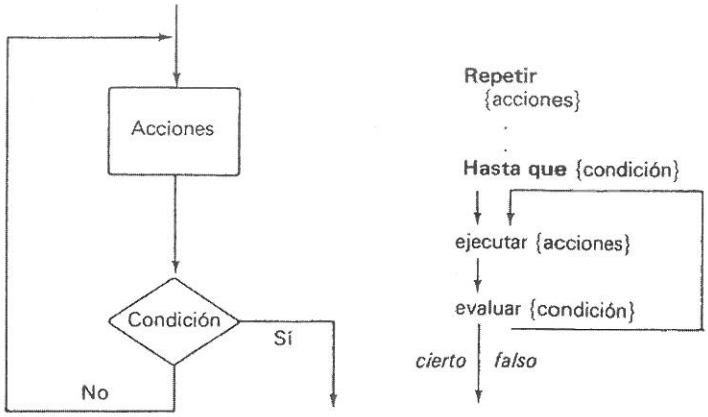
Esta estructura se conoce normalmente como DOWHILE.



3.3.2 Estructura repetir-hasta.

Existe otra estructura en la que el número de iteraciones o repeticiones del grupo de instrucciones se ejecuta hasta que la condición deje de cumplirse. Esta condición se cumple al final.

Esta estructura se conoce también como DOUNTIL.



3.3.3 Estructura para.

La estructura para es aquella que se repite un número fijo de veces.

Algoritmo

para {variable} **de** valor **i a** valor **f** incremento inc
hacer {acciones}

fin_para

El incremento puede ser positivo como el caso anterior o negativo.

para {variable} **de** valor **i a** valor **i a** valor **f** incremento dec
hacer {acciones}

fin_para

4 Recursos abstractos.

La estructuración debe cumplir el uso de recursos abstractos. El proceso de realización de diferentes pasos hasta encontrar la solución de un problema es un proceso abstracto.

Diseñar o concebir un problema en términos abstractos consiste en no tener en cuenta la máquina que lo va a resolver, así como el lenguaje de programación que se va a utilizar. Esto lleva consigo la obtención de un conjunto de acciones que se han de realizar para obtener una solución.

Al considerar un algoritmo y los cálculos que incluyen, se hace abstracción de los valores específicos. Igualmente el concepto de variable implica una abstracción cuando se da un nombre a una operación determinada, y se utiliza considerando lo que hace, pero sin preocuparnos de cómo lo hace.

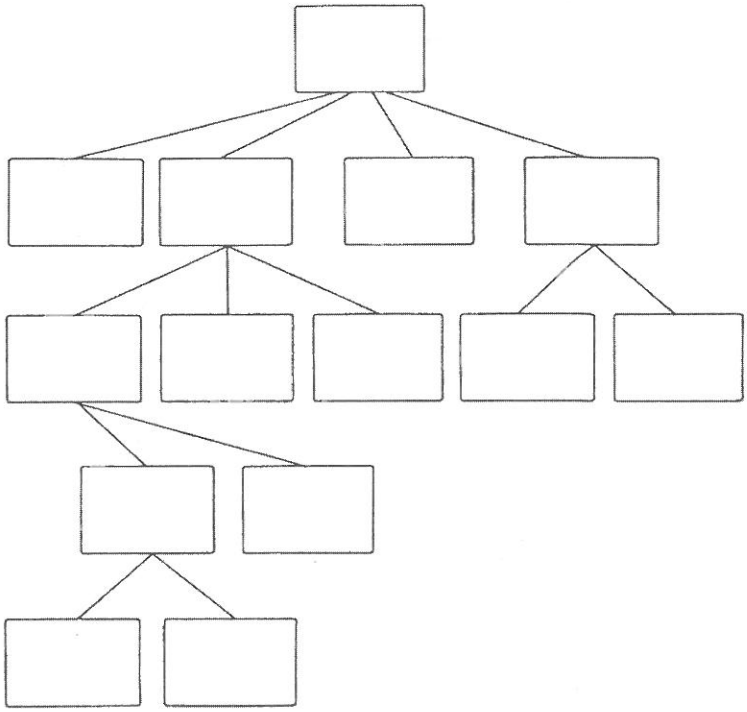
Tras encontrar la solución adecuada mediante el algoritmo, se analiza esta solución con la computadora y el lenguaje de programación que se van a utilizar, a fin de comprobar si las diferentes acciones son susceptibles de ser ejecutadas por la máquina tal y como han sido concebidas; si eso no fuera posible, será preciso descomponer las acciones en otras subacciones más elementales, continuándose el proceso hasta que cada subacción pueda ser codificada en el lenguaje elegido y por consiguiente ejecutado en la computadora del usuario.

5 Metodología descendente "top-down".

La metodología o diseño descendente (*top-down*), consiste en establecer una serie de niveles de menor a mayor complejidad que den solución al problema. En esencia, consiste en efectuar una relación entre etapas de la estructuración, de forma que una etapa jerárquica y su inmediatamente inferior se relacionen mediante entradas y salidas de información.

Un programa estructurado tiene una representación en forma de árbol.

La marcha analítica de un proceso descendente estaría basada en dos características esenciales: representación en forma de árbol y descomposición funcional. El diseño se basa en la realización de diferentes niveles. El primer nivel resuelve totalmente el problema y el segundo y sucesivos niveles son refinamientos sucesivos del primero (*stepwise*) y se sigue siempre la metodología de recursos abstractos. Si el diseño y planteamiento es correcto, nunca será preciso volver atrás, ya que los niveles anteriores al que se esté situado en un momento dado ya habrán resuelto el problema en su totalidad.



6 Funciones y procedimientos.

6.1 Definición de procedimiento y función.

Un procedimiento está formado por un conjunto de sentencias a las que se asocia un identificador, y que realizan una acción que se reconoce por los cambios que se producen en un conjunto de variables o por realizar alguna operación de entrada/salida. Para realizar su acción el procedimiento puede recibir una serie de valores a través de un conjunto de variables o parámetros que acompañan su definición. Las variables que son modificadas pueden ser variables de las que actúan como parámetros o no serlo.

Una función está constituida por un conjunto de sentencias a las que se asocia un identificador, y cuyo efecto se manifiesta porque producen un valor que es asignado al nombre de la función. Para generar este valor pueden recibir un conjunto de valores a partir de unas variables o parámetros que se especifican en su declaración.

Una vez definida una función o un procedimiento podemos usarlo en cualquier parte del programa donde necesitemos realizar la acción que desarrolla, con tan solo invocarlo mediante su nombre.

Las dos posibles visiones que podemos ofrecer de ellos son la abstracción operacional y la transferencia de control.

6.1.1 Abstracción operacional.

Esta es la que corresponde a la descripción que acabamos de ofrecer de los procedimientos. El programa le pasa unos valores de entrada y el procedimiento realiza una acción que consiste en modificar el estado de ciertas variables, o en el caso de las funciones que consiste en devolver un valor.

Por tanto, para usarlos sólo es necesario conocer qué valores es necesario suministrarle y conocer cómo manifiesta su efecto, es decir saber si es una función o un

procedimiento y en este caso conocer las variables que modifica. Pero no nos interesa el cómo hace la operación.

6.1.2 Visión de transferencia de control.

Cuando la ejecución de un programa llega a una sentencia de invocación de un procedimiento, el control se transfiere hacia la primera sentencia de éste. Cuando el procedimiento llega a su final, el control se transfiere a la sentencia que sigue a aquella donde se produjo la invocación.

6.2 Elementos básicos.

- Identificador. Nombre que sirve para invocarlo desde cualquier parte del programa.
- Lista de parámetros. Los parámetros son una lista de cero o varias variables que permiten la comunicación entre un procedimiento y el programa que lo usa (que también puede ser otro procedimiento). Cuando son declarados se especifica esta lista y cuando son invocados es necesario que una lista de argumentos acompañe al nombre. Se realiza una correspondencia uno a uno entre parámetro y argumento. Los argumentos son expresiones del mismo tipo de dato que el parámetro que aparece en la misma posición en la lista de parámetros. Estos permiten construir procedimientos generales, ya que es posible definir una operación y en el momento de la invocación establecer sobre qué objetos de datos se realiza. Los parámetros pueden ser de dos tipos:
  - Parámetros formales: variables locales de un subprograma utilizadas para la recepción y el envío de los datos. Son siempre fijos.
  - Parámetros actuales: variables y datos enviados, en cada llamada de subprograma, por el programa o subprograma llamante. pueden ser cambiados para cada llamada.
- Cuerpo. El cuerpo es el conjunto de sentencias que corresponden a la especificación de la operación.
- Entorno. Es el conjunto de variables externas al procedimiento que son accesibles y pueden ser modificadas o simplemente usadas.

6.3 Paso de parámetros.

El proceso de emisión y recepción de datos y resultados mediante variables de enlace se denomina paso de parámetros. El paso de parámetros puede realizarse de dos formas distintas

- Paso por valor: para suministrar datos de entrada al subprograma.
- Paso por referencia: para entrada y salida o sólo salida.

Un parámetro actual pasado por valor es un dato, o una variable global que contiene un dato de entrada para el subprograma. Esta variable no puede ser modificada por el subprograma, que copia su valor en el parámetro formal correspondiente para poder utilizarlo.

Un parámetro actual pasado por referencia es una variable del programa o subprograma llamante, que puede contener o no un dato para el subprograma llamado,

el cual coloca un resultado en esa variable, que queda a disposición del llamante una vez concluida la ejecución del subprograma.

Los parámetros formales se comportan como variables locales, con la particularidad de que en cada llamada del subprograma se identifican con los parámetros actuales, según su colocación; esto es, cada parámetro actual, si es pasado por valor, se copia en el parámetro formal correspondiente, y si es pasado por referencia, proporciona su dirección de memoria al parámetro formal asociado.

Es de destacar el hecho de que, desde el punto de vista físico, en el paso por valor no se proporciona la variable al subprograma, sino solamente su contenido, evitando así su modificación, y en el paso por referencia se proporciona la dirección o referencia de la variable, con lo que el subprograma la utiliza como propia, modificándola si es necesario, para dejar en ella los resultados que ha de devolver.

La utilización de parámetros por referencia supone ahorro de memoria, puesto que la variable local correspondiente no existe físicamente, sino que se asocia a la global en cada llamada. También supone el riesgo de modificar por error una variable global sin desearlo.

6.4 Grado de entrada, grado de salida, visibilidad, y conectividad.

El grado de entrada (*fan-in*) de un módulo es el número de módulos que le llaman directamente. Un fan-in elevado es el resultado una factorización (separación de una función contenida como código en un módulo, como un nuevo módulo) inteligente y de la eliminación de módulos restrictivos. A la hora de programar, el hecho de tener una función llamada por diferentes módulos evita la necesidad de codificar prácticamente la misma función en distintos lugares. No debe preocupar el hecho de que las llamadas se produzcan desde módulos de diferentes niveles. Si el módulo es verdaderamente útil puede utilizarse por cualquier otro módulo del sistema. Sin embargo, hay que tener en cuenta dos reglas que restringen el uso del fan-in:

- Los módulos con fan-in deben tener una buena cohesión.
- Cada interfaz hacia un módulo sencillo debe tener el mismo número y tipo de parámetros.

El grado de salida (*fan-out*) de un módulo es el número de módulos inmediatamente subordinados de ese módulo. Se ha de intentar limitar el fan-out de un módulo. Un módulo con demasiados subordinados puede remediarse mediante la factorización.

La visibilidad (*scope*) indica el conjunto de componentes del programa que pueden ser invocados o utilizados sus datos por un componente dado, incluso cuando se haga indirectamente. Por ejemplo, un módulo en un sistema orientado a los objetos puede tener acceso a muchos objetos de los que herede, pero puede que sólo utilice unos pocos de esos objetos. Todos los objetos son visibles para el módulo.

La conectividad indica el conjunto de componentes a los que directamente se invoca o se utilizan sus datos en un determinado módulo. Por ejemplo, un módulo que directamente puede provocar la ejecución de otro módulo, está conectado a ese último.

6.5 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 usuario. Así, por ejemplo, $f(x)=x/(1+x^2)$, donde f es el nombre de la función y x es el argumento. Observemos que ningún valor específico se asocia con x; es un parámetro formal utilizado en la definición de la función. Para evaluar f debemos darle un valor real o actual a x, con este valor se puede calcular el resultado.

Una función puede tener varios argumentos, sin embargo, solamente un único valor se asocia con la función para cualquier conjunto dado de argumentos.

Cada lenguaje de programación tiene sus propias funciones incorporadas que se utilizan escribiendo sus nombres con los argumentos adecuados. Cada función se evoca utilizando su nombre en una expresión con los argumentos actuales o reales encerrados entre paréntesis.

Las funciones incorporadas al sistema se denominan funciones internas o intrínsecas y las funciones definidas por usuario 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 le llama explícitamente sino que se le invoca o referencia mediante un nombre y una lista de parámetros actuales.

El algoritmo o programa llama o invoca a la función con el nombre de esta última 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.

Las funciones son diseñadas para realizar tareas específicas: tomar una lista de valores (llamados argumentos) y devolver un único valor.

6.5.1 Declaración de funciones.

La declaración de una función requiere una serie de pasos que la definen. Una función como tal subalgoritmo o subprograma tiene una constitución similar a los algoritmos; por consiguiente, constará de una cabecera con la definición de la función y seguido por el cuerpo de la función, que serán una serie de acciones o instrucciones cuya ejecución hará que se asigne un valor al nombre de la función. Esto designa el valor particular del resultado que ha de devolverse al programa llamador.

La declaración de la función será:

```
función nombre_función (par1, par2, par3,...)
  <acciones>
```

par1, par2...	lista de *parámetros formales* o *argumentos*
nombre_función	nombre asociado con la función, que será un nombre de identificador válido
<acciones>	instrucciones que constituyen la definición de la función, y que debe contener una acción sola de asignación que asigne un valor al nombre de la función, es decir: nombre_función ← expresión

Para que las acciones descritas en un subprograma función sean ejecutadas, se necesita que éste sea invocado desde un programa principal o desde otros subprogramas a fin de proporcionarle los argumentos de entrada necesarios para realizar esas acciones.

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

6.5.2 Invocación a las funciones.

Una función puede ser llamada sólo mediante referencia de la forma siguiente:

nombre_función *(lista de parámetros actuales)*

nombre función	función que llama
lista de parámetros actuales	constantes, variables, expresiones, valores de funciones, nombres de funciones o procedimientos

Cada vez que se llama a una función desde el algoritmo 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 (esta correspondencia se denomina llamada por valor).
2. Se ejecuta el cuerpo de acciones de la función.
3. Se devuelve el valor de la función y se retorna al punto de llamada.

Las funciones pueden tener muchos argumentos, pero solamente un resultado: el valor de la función. Esto limita su uso, aunque se encuentran con frecuencia en cálculos científicos.

6.6 Procedimientos (subrutinas).

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 la función no es apropiada y se necesita disponer del otro tipo de subprograma: el procedimiento o subrutina.

Un procedimiento o subrutina 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, por ejemplo SORT, para indicar que un procedimiento denominado SORT se va a usar. Cuando se invoca el procedimiento, los pasos que lo definen se ejecutan y a continuación se devuelve el control al programa que le llamó.

6.6.1 Declaración de procedimientos.

La declaración de un procedimiento es similar a la de funciones.

```
procedimiento nombre (parámetros formales; parámetros variables)
        <acciones>
```

Los parámetros formales tienen el mismo significado que en las funciones; los parámetros variables (en aquellos lenguajes que los soportan) 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.

6.6.2 Llamadas a procedimientos.

El procedimiento se llama mediante la instrucción:

```
[llamar_a] nombre (lista de parámetros actuales)
```

La palabra llamar-a (*call*) es opcional y su existencia depende del lenguaje de programación.

6.7 Sustitución de argumentos/parámetros.

La lista de parámetros, bien formales en el procedimiento o actuales (reales) en la llamada, se conocen como lista de parámetros. Cuando se llama al procedimiento, cada parámetro formal se sustituye por el correspondiente parámetro actual.

Las acciones sucesivas a realizar son las siguientes:

1. Los parámetros reales son sustituidos por los parámetros formales.
2. El cuerpo de la declaración del procedimiento se sustituye por la llamada del procedimiento.
3. Por último, se ejecutan las acciones escritas por el código resultante.

6.8 Ámbito: variables locales y globales.

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

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 se confina al procedimiento en el que está declarada. Cuando otro subprograma utiliza el mismo nombre se refiere a una posición diferente 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 o algoritmo completo.

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

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 sólo necesitamos conocer lo que hace y no tenemos que estar preocupados por su diseño, es decir, como están programados.

Esta característica hace posible dividir grandes proyectos en piezas más pequeñas independientes. Cuando diferentes programadores están implicados, ellos pueden trabajar independientemente.

A pesar del hecho importante de los subprogramas independientes y las variables locales, la mayoría de los lenguajes proporcionan algún método para tratar ambos tipos de variables.

Una variable local a un subprograma no tiene ningún significado en otros subprogramas. Si un subprograma asigna un valor a una de sus variables locales, este valor no es accesible a otros subprogramas, 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.

En un programa sencillo con un subprograma cada variable u otro identificador es o bien local al procedimiento o global al programa completo. Sin embargo, si el programa incluye procedimientos que llaman a otros procedimientos (procedimientos anidados), entonces la noción de global/local es algo más complicado de entender.

El ámbito de un identificador (variables, constantes, procedimientos) es la parte del programa donde se conoce el identificador. Si un procedimiento está definido localmente a otro procedimiento, tendrá significado sólo dentro del ámbito de ese procedimiento. A las variables les sucede lo mismo; si están definidas localmente dentro de un procedimiento, su significado o uso se confina a cualquier función o procedimiento que pertenezca a esa definición.

Los lenguajes que admiten variables locales y globales suelen tener la posibilidad explícita de definir dichas variables como tales en el cuerpo del programa, o, lo que es lo mismo, definir su ámbito de actuación. Para ello se utilizan las cabeceras de programas y subprogramas, con lo que se definen los ámbitos. Las variables definidas en un ámbito son accesibles en él, es decir, en todos los procedimientos interiores.

6.9 Procedimiento vs. función.

Los procedimientos y funciones son subprogramas cuyo diseño y misión son similares; sin embargo, existen unas diferencias esenciales entre ellos:

1. Un procedimiento es llamado desde el algoritmo o programa principal mediante su nombre y una lista de parámetros actuales, o bien con la instrucción llamar (*call*). Al llamar al procedimiento se detiene momentáneamente el programa que se estuviera realizando y el control pasa al procedimiento llamado. Después que las acciones del procedimiento se ejecutan, se regresa a la acción inmediatamente siguiente a la que se llamó.
2. Las funciones devuelven un valor, las subrutinas pueden devolver 0, 1 ó n valores y en forma de lista de parámetros.
3. El procedimiento se declara igual que la función, pero su nombre no está asociado a ninguno de los resultados que obtiene.

7 Conclusiones.

La programación no debe considerarse como un arte sino como una actividad con fundamentos lógicos y formales, cuyo objetivo principal es crear programas que correspondan a la especificación del problema planteada inicialmente.

La programación estructurada es una técnica de construcción de programas que utiliza al máximo los recursos del lenguaje, limita el conjunto de estructuras aplicables a leer y presenta una serie de reglas que coordinan adecuadamente el desarrollo de las diferentes fases de la programación. Para ello, utiliza en su diseño los siguientes conceptos o principios fundamentales: estructuras básicas, recursos abstractos, y diseño descendente (*top-down*).

Las estructuras básicas son la estructura secuencial; la estructura alternativa, que incluye la estructura alternativa simple, la estructura alternativa doble, y la estructura alternativa múltiple; las estructuras repetitivas, entre las que se encuentran la estructura mientras, la estructura repetir-hasta, y la estructura para.

La estructuración debe cumplir el uso de recursos abstractos que consiste en no tener en cuenta la máquina que lo va a resolver, así como el lenguaje de programación que se va a utilizar. Esto lleva consigo la obtención de un conjunto de acciones que se han de realizar para obtener una solución.

La metodología o diseño descendente (*top-down*), consiste en establecer una serie de niveles de menor a mayor complejidad que den solución al problema. Consiste en efectuar una relación entre etapas de la estructuración, de forma que una etapa jerárquica y su inmediatamente inferior se relacionen mediante entradas y salidas de información.

El diseño descendente (*top-down*) normalmente dará lugar a subproblemas que suelen resolverse mediante funciones y/o procedimientos. Un procedimiento está formado por un conjunto de sentencias a las que se asocia un identificador, y que realizan una acción que se reconoce por los cambios que se producen en un conjunto de variables o por realizar alguna operación de entrada/salida. Una función está constituida por un conjunto de sentencias a las que se asocia un identificador, y cuyo efecto se manifiesta porque producen un valor que es asignado al nombre de la función.