RECURSIÓN (RECURSIVIDAD)

Como ya se conoce, un subprograma puede llamar a cualquier otro subprograma y éste a otro, y así sucesivamente; dicho de otro modo, los subprogramas se pueden anidar. Se puede tener

A llamar_a B, B llamar_a C, C llamar_a D

Cuando se produce el retorno de los subprogramas a la terminación de cada uno de ellos el proceso resultante será

D retornar_a C, C retornar_a B, B retornar_a A

¿Qué sucedería si dos subprogramas de una secuencia son los mismos?

A llamar_a A

o bien

A llamar_a B, B llamar_a A

En primera instancia, parece incorrecta. Sin embargo, existen lenguajes de programación, C, entre otros— en que un subprograma puede llamarse a sí mismo. Una función o procedimiento que se puede llamar a sí mismo se llama recursivo. La recursión (recursividad) es una herramienta muy potente en algunas aplicaciones, sobre todo de cálculo. La recursión puede ser utilizada como una alternativa a la repetición o estructura repetitiva. El uso de la recursión es particularmente idóneo para la solución de aquellos problemas que pueden definirse de modo natural en términos recursivos. La escritura de un procedimiento o función recursiva es similar a sus homónimos no recursivos; sin embargo, para evitar que la recursión continúe indefinidamente es preciso incluir una condición de terminación. La razón de que existan lenguajes que admiten la recursividad se debe a la existencia de estructuras específicas tipo pilas (stack, en inglés) para este tipo de procesos y memorias dinámicas. Las direcciones de retorno y el estado de cada subprograma se guardan en estructuras tipo pilas.

LA NATURALEZA DE LA RECURSIVIDAD

En algunos problemas es útil disponer de funciones que se llamen a sí misma. Un subprograma recursivo es un subprograma que se llama a sí mismo ya sea directa o indirectamente. La recursividad es un tópico importante examinado frecuentemente en cursos de programación y de introducción a las ciencias de la computación. En matemáticas existen numerosas funciones que tienen carácter recursivo; de igual modo numerosas circunstancias y situaciones de la vida ordinaria tienen carácter recursivo. Hasta el momento casi siempre se han visto subprogramas que llaman a otros subprogramas distintos. Así, si se dispone de dos procedimientos proc1 y proc2, la organización de un programa tal y como se suele haber visto hasta este momento podría adoptar una forma similar a esta:

procedimiento proc1(...)

inicio

...

fin_procedimiento

procedimiento proc2(...)

inicio

...

proc1(...) // llamada a proc1

...

fin_procedimiento

Cuando diseñan programas recursivos se tendría esta situación:

procedimiento proc1(...)

inicio

...

proc1(...);

...

fin_procedimiento

o bien esta otra:

procedimiento proc1(...)

inicio

...

proc2(...) //llamada a proc2

...

fin_procedimiento

procedimiento proc2(...)

inicio ...

proc1(...) //llamada a proc1 ...

fin_procedimiento

RECURSIVIDAD DIRECTA E INDIRECTA

En recursión directa el código del subprograma recursivo F contiene una sentencia que invoca a F, mientras que en recursión indirecta el subprograma F invoca al subprograma G que invoca a su vez al subprograma P, y así sucesivamente hasta que se invoca de nuevo al subprograma F.

Si una función, procedimiento o método se invoca a sí misma, el proceso se denomina recursión directa; si una función, procedimiento o método puede invocar a una segunda función, procedimiento o método que a su vez invoca a la primera, este proceso se conoce como recursión indirecta o mutua.

Un requisito para que un algoritmo recursivo sea correcto es que no genere una secuencia infinita de llamadas sobre sí mismo. Cualquier algoritmo que genere una secuencia de este tipo no puede terminar nunca. En consecuencia, la definición recursiva debe incluir un componente base (condición de salida) en el que f(n) se defina directamente (es decir, no recursivamente) para uno o más valores de n. Debe existir una "forma de salir" de la secuencia de llamadas recursivas. Así en la función f(n) = n! para n entero

$$f(n)\begin{cases} 1 & n \leq 1 \\ n*f(n-1) & n > 1 \end{cases}$$

la condición de salida o base es f(n) = 1 para n ≤ 1.

En el caso de la serie de Fibonacci

F0 = 0, F1 = 1, Fn = Fn-1 + Fn-2 para n > 1.

F0 = 0 y F1 = 1 constituyen el componente base o condiciones de salida y Fn = Fn-1 + Fn-2 es el componente recursivo.

Recursividad indirecta

La recursividad indirecta se produce cuando un subprograma llama a otro, que eventualmente terminará llamando de nuevo al primero. El programa principal llama a la función recursiva A() con el argumento 'Z' (la última letra del alfabeto). La función A examina su parámetro c. Si c está en orden alfabético después que 'A', la función llama a B(), que inmediatamente llama a A(), pasándole un parámetro predecesor de c. Esta acción hace que A() vuelva a examinar c, y nuevamente una llamada a B(), hasta que c sea igual a 'A'. En este momento, la recursión termina ejecutando putchar() veintiséis veces y visualizando el alfabeto, carácter a carácter.

Condición de terminación de la recursión

Cuando se implementa un subprograma recursivo será preciso considerar una condición de terminación, ya que en caso contrario el subprograma continuaría indefinidamente llamándose a sí mismo y llegaría un momento en que la memoria se podría agotar. En consecuencia, sería necesario establecer en cualquier subprograma recursivo la condición de parada que termine las llamadas recursivas y evitar indefinidamente las llamadas.

RECURSIÓN VERSUS ITERACIÓN

Tanto la iteración como la recursión se basan en una estructura de control: la iteración utiliza una estructura repetitiva y la recursión utiliza una estructura de selección. La iteración y la recursión implican ambas repetición: la iteración utiliza explícitamente una estructura repetitiva mientras que la recursión consigue la repetición mediante llamadas repetidas. La iteración y recursión implican cada una un test de terminación (condición de salida). La iteración termina cuando la condición del bucle no se cumple mientras que la recursión termina cuando se reconoce un caso base o la condición de salida se alcanza.

La recursión tiene muchas desventajas. Se invoca repetidamente al mecanismo de recursividad y en consecuencia se necesita tiempo suplementario para realizar las mencionadas llamadas. Esta característica puede resultar cara en tiempo de procesador y espacio de memoria. Cada llamada de una función recursiva produce que otra copia de la función (realmente sólo las variables de función) sea creada; esto puede consumir memoria considerable. Por el contrario, la iteración se produce dentro de una función, de mo do que las operaciones suplementarias de las llamadas a la función y asignación de memoria adicional son omitidas.

En consecuencia, ¿cuáles son las razones para elegir la recursión? La razón fundamental es que existen numerosos problemas complejos que poseen naturaleza recursiva y, en consecuencia, son más fáciles de implementar con algoritmos de este tipo. Sin embargo, en condiciones críticas de tiempo y de memoria, es decir, cuando el consumo de tiempo y memoria sean decisivos o concluyentes para la resolución del problema, la solución a elegir debe ser, normalmente, la iterativa.

RECURSIÓN INFINITA

La iteración y la recursión pueden producirse infinitamente. Un bucle infinito ocurre si la prueba o test de continuación de bucle nunca se vuelve falsa; una recursión infinita ocurre si la etapa de recursión no reduce el problema en cada ocasión de modo que converja sobre el caso base o condición de salida. En realidad la recursión infinita significa que cada llamada recursiva produce otra llamada recursiva y ésta a su vez otra llamada recursiva y así para siempre. En la práctica dicho código se ejecutará hasta que la computadora agota la memoria disponible y se produzca una terminación anormal del programa. El flujo de control de un algoritmo recursivo requiere tres condiciones para una terminación normal:

• Un test para detener (o continuar) la recursión (condición de salida o caso base).

• Una llamada recursiva (para continuar la recursión).

• Un caso final para terminar la recursión

Bibliografía

Joyanes, L. Fundamentos de Programación. Algoritmos y estructura de datos. McGraw-Hill. México. 1990.