Recursividad

Introducción

En la lección anterior se explicaron los conceptos de función y subrutina; vimos cómo los subprogramas pueden ser invocados desde cualquier parte de un algoritmo y, por tanto, también desde el interior de otra función o subrutina. Si la invocación de un subprograma (función o subrutina) se produce desde el propio subprograma se dice que se trata de un subprograma recursivo.

La recursividad es muy útil para resolver problemas que son definibles en función de sus propios términos o en función de varios casos más sencillos; así, la invocación a un subprograma recursivo supondrá una o más invocaciones que, a su vez, resultarán en nuevas invocaciones. Para que este proceso finalice y pueda retornarse desde la llamada inicial es necesario que exista algún caso en que el subprograma recursivo finalice sin necesidad de invocarse a sí mismo, es decir, un caso base.

Al alumno probablemente le sea más sencillo comprender el concepto de recursividad si se asimila al principio de inducción matemática; como recordará, dicho principio determina que una proposición es válida para todo número natural n si:

1. Es válida para n=1
2. De su validez para un número natural cualquiera n = k se desprende su validez para n = k +1

Como se puede ver, en dicho principio existe un caso base [1] y un caso “recursivo” [2]; de forma análoga se comportan las funciones y subrutinas recursivas, sin embargo, debe quedar claro que esta analogía tiene una finalidad puramente didáctica puesto que son conceptos diferentes.

Hasta este momento el único mecanismo que hemos visto para llevar a cabo acciones repetitivas es la iteración, la recursividad permite resolver los mismos problemas que la iteratividad; así pues, ¿qué opción sería la más adecuada? Respuesta: depende...

En primer lugar debe quedar claro que no existen problemas iterativos y problemas recursivos, cualquier proceso que se pueda resolver de forma iterativa admite una solución recursiva y viceversa; lo que sí es cierto que las soluciones recursivas son más compactas que las iterativas aunque éstas son más eficientes y consumen menos recursos. En definitiva, siempre es posible plantear una solución recursiva en la fase de diseño y transformarla en una solución iterativa en la fase de implementación si se comprueba que la solución iterativa va a ser mucho más eficiente que la versión recursiva y no excesivamente compleja.

Diseño de algoritmos recursivos

Para desarrollar algoritmos recursivos siempre debe tenerse en cuenta que ya existe un algoritmo que permite resolver una versión reducida del problema; con esto en mente se procede como sigue:

1. Identificar subproblemas atómicos (no descomponibles); estos subproblemas deben resolverse de forma directa, nunca utilizando el algoritmo general puesto que son los ya conocidos casos base.
2. Descomponer el problema en subproblemas; estos subproblemas se resuelven apoyándose en el algoritmo que sabemos ya existe; la solución de estos subproblemas debe acercarnos de forma progresiva al caso base.
3. Probar de manera informal que tanto los casos base como los casos generales encuentran solución aplicando el algoritmo diseñado.

A continuación se presentan algunos ejemplos de problemas que pueden resolverse de forma sencilla aplicando recursividad.

Factorial de un número

Supongamos que deseamos diseñar un algoritmo que permita calcular el factorial de un número n, n!, para ello aplicaremos el “método” de diseño anteriormente explicado.

En primer lugar, partimos del supuesto de que existe un algoritmo que permite resolver el problema del factorial de un número en una versión más sencilla que *n!*, por ejemplo *(n-1)!*

Con ese conocimiento en mente, determinamos el caso o casos base que no precisan de dicho algoritmo más sencillo para su resolución:

$0! = 1$

A continuación, descomponemos el problema en subproblemas resolubles mediante la aplicación del algoritmo ya existente:

$n! = n \cdot (n-1)!$

Comprobamos que las dos expresiones anteriores permiten calcular *0!*, *1!*, *2!*, *4!*, *10!*, etc. Podemos expresar este algoritmo para calcular factoriales de números de manera informal:

```
si n es 0 entonces
  factorial ← 1
si no
  factorial ← n · (n-1)!
```

Potencia de un número

Supongamos que deseamos diseñar un algoritmo que permita calcular la potencia n-sima de un número *a*, a^n, para obtener una versión recursiva de dicho algoritmo se procede de manera similar al caso anterior.

Suponemos que ya existe un algoritmo que permite resolver una forma más sencilla del problema que nos ocupa, por ejemplo, a^{n-1}. Sabiendo esto, es posible determinar el caso o casos base, aquellos subproblemas con una solución directa:

$a^0 = 1$

Se descompone entonces el problema en subproblemas resolubles mediante ese algoritmo:

$a^n = a \cdot a^{n-1}$

Se comprueba que las dos expresiones anteriores permiten calcular 2^0, 4^3, 10^5, 3^3, etc. Una descripción no formalizada del algoritmo completo sería la siguiente:

```
si n es 0 entonces
  potencia ← 1
si no
  potencia ← a · an-1
```

Sintáxis de subprogramas recursivos en la notación algorítmica

La sintáxis para describir funciones y subrutinas recursivas en la notación algorítmica que utilizamos es muy simple; como sabemos un subprograma es recursivo si se invoca a sí mismo, así pues un ejemplo de subrutina recursiva sería el siguiente:

```
accion nombre_subrutina ([ent|sal|ent sal] arg1∈tipo1, ..., [ent|sal|ent sal] argN∈tipoN)
inicio
  si es caso base
    sentencia1
    sentencia2
    ...
    sentenciaN
  si no
    sentencia1
    sentencia2
    ...
    llamar nombre_subrutina (a1, ..., aN)
    ...
    sentenciaN
  fin si
fin
```

Mientras que en el caso de una función recursiva sería como sigue:

```
tipo funcion nombre_funcion (arg1 ∈ tipo1, arg2 ∈ tipo2, ..., argN ∈ tipoN)
inicio
  si es caso base
    sentencia1
    sentencia2
    ...
    sentenciaN
    nombre_funcion ← expresión
  si no
    sentencia1
    sentencia2
    ...
    sentenciaN
    nombre_funcion ← nombre_funcion (a1, a2, ..., aN)
  fin si
fin
```

Obsérvese que en ambos casos se distingue perfectamente el momento en que el subprograma se encuentra en un caso base (que no requiere una invocación recursiva) y cuándo en un caso general; esto es muy importante puesto que en caso de que no se diseñe de forma adecuada, el algoritmo entrará en un bucle infinito.

A continuación se muestran los anteriores algoritmos para calcular el factorial y la potencia de un número formalizados de acuerdo con esta notación, dado que se trata de subprogramas que deben retornar un único valor se implementarán como funciones.

```
entero funcion factorial (n ∈ entero)
inicio
  si n=0 entonces
    factorial ← 1
  si no
    factorial ← n · factorial(n-1)
  fin si
fin
```

La función que calcula la potencia an:

```
entero funcion potencia (a, n ∈ entero)
inicio
  si n=0 entonces
    potencia ← 1
  si no
    potencia ← a · potencia(a, n-1)
  fin si
fin
```

Nótese que ninguna de las funciones anteriores lleva a cabo las necesarias validaciones, se ha hecho a sí a fin de presentar un par de ejemplos claros de funciones recursivas; las siguientes son las versiones con validación (devuelven un valor absurdo a fin de señalar el error):

```
entero funcion factorial (n ∈ entero)
inicio
  si n>=0 entonces
    si n=0 entonces
      factorial ← 1
    si no
      factorial ← n · factorial(n-1)
    fin si
  si no
    factorial ← -1
  fin si
fin

entero funcion potencia (a, n ∈ entero)
inicio
  si n>=0 entonces
    si n=0 entonces
      potencia ← 1
    si no
      potencia ← a · potencia(a, n-1)
    fin si
  si no
    potencia ← 0
  fin si
fin
```

Sintáxis de subprogramas recursivos en FORTRAN 90

FORTRAN 77 no soporta recursividad y FORTRAN 90 lo hace pero de una manera un tanto artificiosa; en primer lugar no basta con construir una subrutina o función recursiva, es necesario indicarlo explícitamente mediante la palabra reservada RECURSIVE:

```
recursive subroutine nombre_subrutina (arg1, ..., argN)
  declaración arg1
  ...
  declaración argN

  if caso base then
    sentencia1
    sentencia2
    ...
    sentenciaN
  else
    sentencia1
    sentencia2
    ...
    call nombre_subrutina(a1, ..., aN)
    ...
    sentenciaN
  end if
end subroutine
```

Por lo que se refiere a las funciones no sólo hay que indicar que son recursivas mediante el uso de `RECURSIVE` sino que también es necesario indicar una variable de retorno puesto que FORTRAN 90 no admite que el identificador de la función aparezca simultaneamente en la parte izquierda y derecha de una asignación.

```
tipo recursive function nombre_funcion (arg1, arg2, ..., argN) result (retorno)
  declaración arg1
  declaración arg2
  ...
  declaración argN

  if caso base then
  sentencia1
    sentencia2
    ...
    sentenciaN
    retorno = valor caso base
  else
    sentencia1
    sentencia2
    ...
    sentenciaN
    retorno = nombre_funcion (a1, a2, aN)
  end if
end function
```

A continuación se presentan tres ejemplos, los dos primeros consisten en la traducción a FORTRAN 90 de las anteriores funciones (factorial y potencia), el último es una demostración de subrutina recursiva.

```
! Función FORTRAN 90 para calcular el factorial de n
!
integer recursive function factorial (n) result (f)
  implicit none

  integer n

  if (n==0) then
    f = 1
  else
    f = n * factorial(n-1)
  end if
end function

! Función FORTRAN 90 para calcular la potencia n-sima de a
!
integer recursive function potencia (a,n) result (p)
  implicit none

  integer a,n

  if (n==0) then
    p = 1
  else
    p = a * potencia(a,n-1)
  end if
end function

! Subrutina recursiva para mostrar n asteriscos
!
recursive subroutine asteriscos (n)
  implicit none

  integer n

  if (n==1) then
    print *, '*'
  else
    print *, '*'
    call asteriscos(n-1)
  end if
end subroutine
```

¿Por qué es menos eficiente y consume más recursos la recursividad?

Para entender la forma en que funciona la recursividad (y por qué consume entonces más recursos y es menos eficiente que una solución iterativa) es necesario comprender la manera en que se resuelven las llamadas a subprogramas (sean éstas recursivas o no).

Al ejecutarse un programa existe una zona de memoria denominada pila donde se almacena información por cada subprograma activo, donde activo quiere decir que ha sido invocado y aún no ha retornado al programa principal. Esta información consiste en:

- Los argumentos de la función o subrutina.
- Las variables locales del subprograma.
- La "dirección" de retorno, es decir el punto del programa que debe pasar a ejecutarse cuando el subprograma retorne.

Cuando un subprograma es invocado toda esa información es apilada en la pila, al finalizar la ejecución del subprograma esa información es retirada de la pila y en caso de una función sustituida por el valor de retorno, pasándose el control a la dirección indicada.

A continuación se muestra un ejemplo sencillo (sin recursividad):

```
01     program prueba
02       implicit none
03
04       character letra
05
06       letra='a'
07       call escribirCaracter (letra)
08
09       letra='b'
10       call escribirCaracter (letra)
11
12       contains
13
14         subroutine escribirCaracter (c)
15           implicit none
16           character c
17
18           print *, c
19         end subroutine
20     end
```

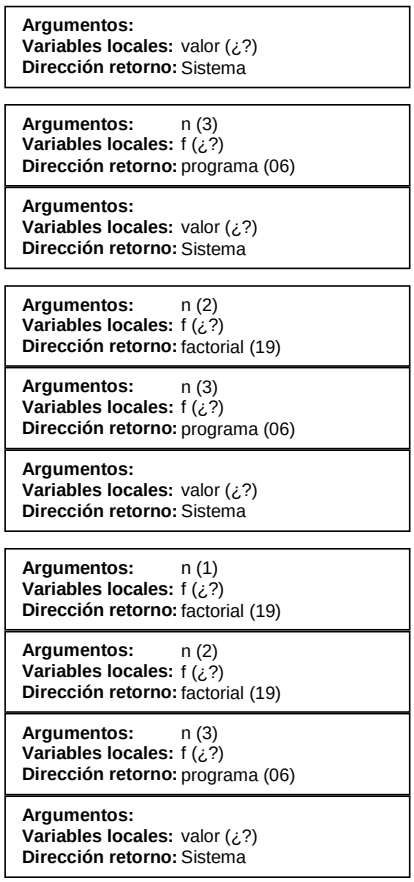
La evolución de la pila en tiempo de ejecución es la siguiente:

Argumentos: **Variables locales:** letra ('a') **Dirección retorno:** Sistema	Al comenzar la ejecución del programa principal en la pila sólo hay información sobre dicho programa; no tiene ningún argumento, sólo variables locales y la dirección de retorno no se encuentra dentro del programa sino que debe retornar al sistema operativo.
Argumentos: c ('a') **Variables locales:** **Dirección retorno:** programa (09) **Argumentos:** **Variables locales:** letra ('a') **Dirección retorno:** Sistema	En la línea `07` se produce la invocación de la subrutina, al pasar a estar activa ésta se debe apilar la información sobre su estado; como se puede ver tiene un argumento, `c`, que toma el valor `'a'` y la dirección de retorno se corresponde con la línea `09` del programa principal.
Argumentos: **Variables locales:** letra ('b') **Dirección retorno:** Sistema	En la línea `09` la pila de ejecución vuelve a tener información referente tan sólo al programa principal; obsérvese como ha cambiado el valor de la variable `letra`.
Argumentos: c ('b') **Variables locales:** **Dirección retorno:** programa (20) **Argumentos:** **Variables locales:** letra ('b') **Dirección retorno:** Sistema	En la línea `10` vuelve a invocarse la subrutina, apilándose información necesaria para su ejecución; nuevamente argumentos y dirección de retorno, en este caso correspondiente a la línea `20` del programa.
Argumentos: **Variables locales:** letra ('b') **Dirección retorno:** Sistema	En la línea `20` se desapila la información de estado de la subrutina y se vuelve a la información sobre el programa principal; cuando éste termine de ejecutarse retornará, como indica su estado, al sistema operativo.

Como se puede ver, por cada invocación de un subprograma es necesario apilar toda una serie de datos y en el momento del retorno desapilarlos; este mecanismo supone una cierta lentitud debido a los movimientos de información hacia y desde la pila además de necesitar un espacio suficiente en caso de que se vayan a producir muchas llamadas encadenadas, como es el caso en un subprograma recursivo.

A continuación se mostrará un ejemplo de programa con una función recursiva y la forma en que evoluciona la pila durante la ejecución de dicho programa.

```
01     program prueba
02       implicit none
03
04       integer valor
05       valor = factorial (3)
06       print *, valor
07
08
09       contains
10
11         integer recursive function factorial (n) result (f)
12           implicit none
13           integer n
14
15           if (n==0) then
16             f = 1
17           else
18             f = n*factorial(n-1)
19           end if
20         end function
21     end
```

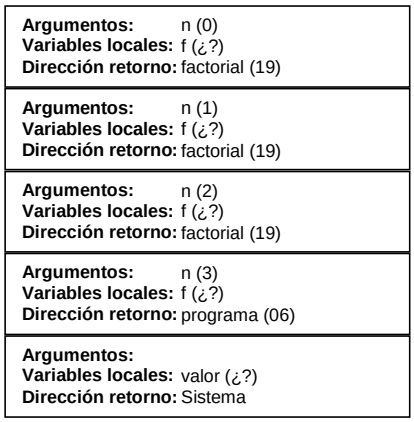


Al comenzar a ejecutarse el programa la única variable del mismo, `valor`, carece de un valor determinado.

En la línea `05` se invoca la función `factorial`, para lo cual se apilan sus argumentos, `3`, y la dirección de retorno. Al ser una función al retornar debe desapilarse esta información y colocarse en la pila el valor de retorno, correspondiente a la línea `06` del programa principal.

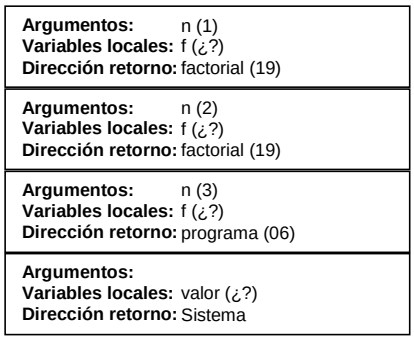
La función `factorial` invocada debe, a su vez, invocarse a sí misma con lo cual debe apilar los argumentos de esa nueva llamada, `2`, y la dirección de retorno que se corresponderá con la línea `19` asociada a la primera llamada de la función.

La segunda función `factorial` invocada debe invocar también otra copia de sí misma, apilando los argumentos, `1`, y la nueva dirección de retorno.



La función `factorial` es invocada, otra vez, y se apilan los datos necesarios, otra vez.

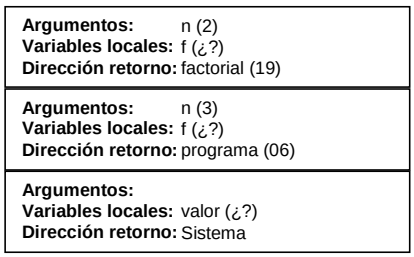
1



Como la última invocación de `factorial` recibe como argumento el `0`, caso base, detiene la serie de invocaciones recursivas y retorna el valor `1` desapilando los datos sobre estado y apilando el valor de retorno.

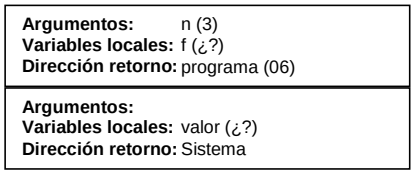
Este valor de retorno es recibido por la penúltima invocación de `factorial`, pudiendo finalizar y retornar, es decir, desapilar su información de estado y apilando su valor de retorno, `1`.

1



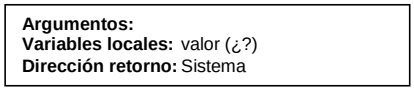
Dicho valor de retorno es recibido por la segunda llamada a `factorial` que puede también finalizar, desapilar su información de estado y apilar su valor de retorno, 2.

2

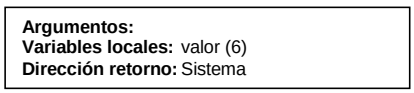


Este valor de retorno es extraído de la pila por la primera invocación de `factorial` que finaliza, desapila su información y coloca en la pila el valor de retorno final de la serie de llamadas recursivas, 6.

6



Este valor de retorno es desapilado por el programa principal y asignado a la variable `valor`.



El programa ha finalizado y puede retornar al sistema operativo.

A la vista del ejemplo está claro que una serie de llamadas recursivas, por corta que resulte, supone un gran número de operaciones con la pila de ejecución lo cual resulta más lento que una serie de operaciones iterativas y, además, requiere un espacio considerable; por esa razón, se dice que la recursividad es más lenta y consume más recursos que una solución iterativa.

Resumen

1. Un subprograma es recursivo cuando se invoca a sí mismo.
2. La recursividad es análoga al principio de inducción y resulta muy útil para resolver problemas definibles en función de sus propios términos.
3. No existen problemas intrínsecamente recursivos o iterativos, todo proceso recursivo puede convertirse en iterativo y viceversa.
4. Los algoritmos recursivos son más simples y compactos que sus correspondientes iterativos, sin embargo su ejecución en un ordenador es más lenta y requiere más recursos.
5. Para diseñar un algoritmo recursivo hay que tener en cuenta lo siguiente:
   - Suponer que ya existe un algoritmo para resolver una versión reducida del problema.
   - Identificar subproblemas atómicos (casos base).
   - Descomponer el problema en subproblemas que se puedan resolver con el algoritmo que sabemos existe y cuya solución vaya aproximándose a los casos base.
   - Probar, de manera informal, que los casos base y los generales encuentran solución con el algoritmo diseñado.
6. La sintáxis de funciones y subrutinas recursivas en la notación algorítmica es inmediata; sin embargo, en FORTRAN 90 resulta un tanto artificiosa:

```
recursive subroutine nombre (arg1, ..., argN)
  declaración arg1
  ...
  declaración argN

  if caso base then
    sentencia1
    sentencia2
    ...
    sentenciaN
  else
    sentencia1
    sentencia2
    ...
    call nombre_subrutina(a1, ..., aN)
    ...
    sentenciaN
  end if
end subroutine
```

```
tipo recursive function nombre (arg1, ..., argN) result (retorno)
  declaración arg1
  declaración arg2
  ...
  declaración argN

  if caso base then
    sentencia1
    sentencia2
    ...
    sentenciaN
    retorno = valor caso base
  else
    sentencia1
    sentencia2
    ...
    sentenciaN
    retorno = nombre_funcion (a1, a2, aN)
  end if
end function
```