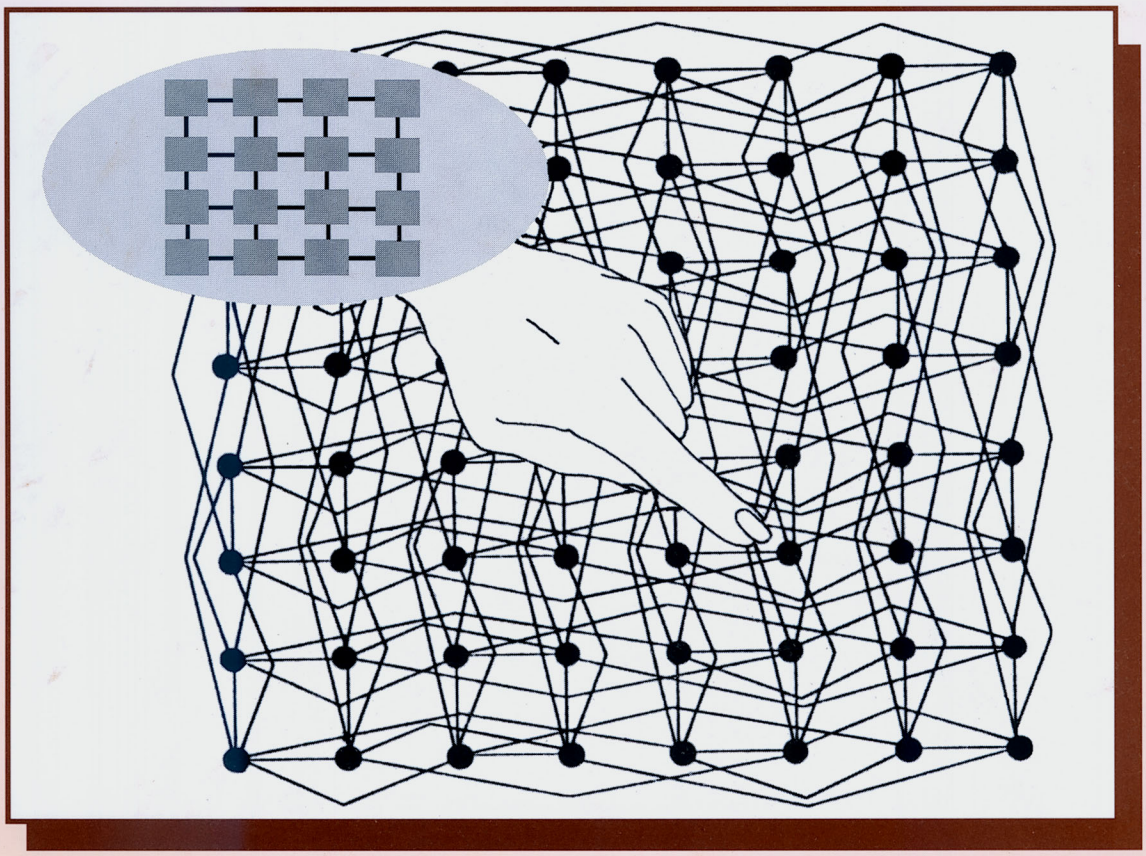
Ingeniería

Revista de la Universidad de Costa Rica
Julio/Diciembre 1996 VOLUMEN 6 Nº 2



INGENIERIA

Revista Semestral de la Universidad de Costa Rica
Volumen 6, Julio/Diciembre 1996 Número 2

CORRESPONDENCIA Y SUSCRIPCIONES

Editorial de la Universidad de Costa Rica
Apartado Postal 75
2060 Ciudad Universitaria Rodrigo Facio
San José, Costa Rica

CANJES

Universidad de Costa Rica
Sistema de Bibliotecas, Documentación e Información
Unidad de Selección y Aquisiciones-CANJE
Ciudad Universitaria Rodrigo Facio
San José, Costa Rica

Suscripción anual:
Costa Rica: ¢ 1 000,00
Otros países: US $ 25,00

Número suelto:
Costa Rica: ¢ 750,00
Otros países: $ 15,00

Edición aprobada por la Comisión Editorial de la Universidad de Costa Rica

Ciudad Universitaria Rodrigo Facio
San José, Costa Rica.

Revisión Filológica: *Lorena Rodríguez*

Diagramación:
José R. Argüello V.

Control de Calidad:
Unidad Diseño Revistas. Oficina de Publicaciones

Impreso en la Oficina de Publicaciones
de la Universidad de Costa Rica

Revista
620.005
I-46i Ingeniería / Universidad de Costa Rica. ——
Vol. I, no. 1 (ene./jun. 1991)— . —— San José, C. R. : Editorial de la Universidad de Costa Rica, 1991— (Oficina de Publicaciones de la Universidad de Costa Rica)
v. : il

Semestral.

1. Ingeniería - Publicaciones periódicas.

CCC/BUCR—250

Ingeniería 6 (2): 31-44 1996 San José, Costa Rica

TRES FORMAS DIFERENTES DE EXPLICAR LA RECURSIVIDAD

Adolfo Di Mare[1]

Resumen

Se presentan tres ejemplos sencillos de programas que ayudan, al programador novato, a entender rápidamente qué es recursividad y cómo funciona .

Summary

Three simple examples are the conduit to lead the novice to an easier learning of recursion and how it works

1. INTRODUCCIÓN.

Desde que fui estudiante siempre he sabido lo difícil que es entender qué es la recursividad y para qué sirve en computación. Para muchos la única forma de aprender a dominar esta importante técnica de programación es correr mentalmente muchos programas recursivos, hasta que, por insistencia, se llega a entender de qué trata este asunto. Como profesor, he tratado de explicar este importante concepto y, con el tiempo, he encontrado las tres formas de hacerlo que expongo aquí.

Como la recursividad es un concepto fundamental en computación, quiero compartir mi experiencia en la enseñanza de esta técnica de programación. Agradeceré cualquier sugerencia que ayude a mejorar este trabajo.
Presento tres enfoques diferentes, cada uno un poco más complejo que el anterior. El primero es una aplicación muy simple de recursividad para crear un comando .bat para el sistema operativo DOS/pc. El segundo es el famoso ejemplo del factorial escrito en Pascal, y el último es el clásico recorrido PID (Proceso-Izquierda-Derecha) para árboles. Cada ejemplo ilustra un componente diferente del concepto de recursión.

1. `ADONDE.bat`: un comando DOS recursivo

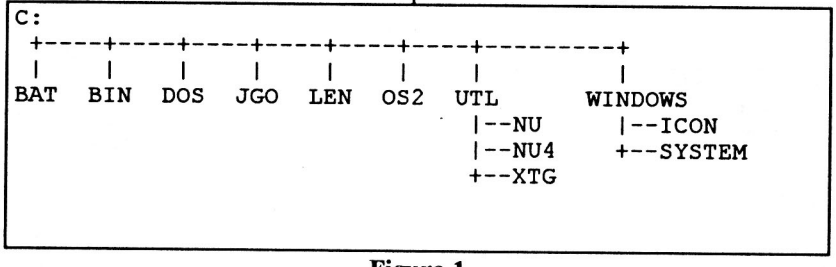


Figura 1

Como los discos duros de los computadores personales son cada vez más grandes, contienen más y más programas. Por ejemplo, en mi disco duro tengo 1,400 archivos y programas diferentes. Para organizar el disco duro, y evitar que los programas aparezcan todos juntos, el sistema operativo DOS permite distribuir los archivos y programas en directorios.

Para organizar los directorios en mi disco duro he usado las sugerencias expuestas en [Arg-91].

[1] Profesor Escuela de Ciencias de la Computación e Informática.
Fac. de Ingeniería. Universidad de Costa Rica.

El resultado es una estructura de directorios como la de la Figura 1. Cada vez que desde la línea de comandos DOS el usuario llama a un programa, DOS lo busca en el directorio actual, y si no lo encuentra, lo busca en la lista de directorios definida por la variable de ambiente PATH. El comando SET sirve para desplegar el valor de todas las variables de ambiente en uso. En mi caso, las variable de ambiente que uso son las que están en la parte superior de la Figura 2. Como programador, tengo una manera de organizar mi disco duro diferente a la de la mayoría de la gente. Otras personas usan un PATH como el de la parte inferior de la Figura 2.

```
PROMPT=$P$G
COMSPEC=C:\BIN\COMMAND.COM
DELDIR=C:\DELETE,2048;
NU=C:\UTL\NU
VDE=C:\BIN\VDE
TMP=E:\TMP
TEMP=E:\TEMP
PATH=C:\UTL;C:\BAT;C:\UTL\NU;C:\UTL\NU4;C:\DOS

PATH=C:\DOS;C:\WINDOWS;C:\WP;C:\LOTUS;C:\FAXIT;C:\XTPRO
```

Figura 2

```
C:\ARTICULO>adonde comm*
c:\bin;e:\tmp
C:\DOS\COMMAND.620
C:\DOS\COMMAND.COM
c:\bin\COMMAND.COM
c:\bin\COMMAND.500
c:\bin\COMMAND.620
c:\bin\COMMAND.622
C:\ARTICULO>adonde adonde
C:\BAT\ADONDE.BAT
```

Figura 3

Independientemente de cómo esté organizado nuestro disco duro, todos necesitamos un PATH para que DOS encuentre los programas que usamos. A veces necesitamos saber en qué directorio se encuentra un programa, y para hacerlo debemos verificar uno a uno los directorios que se mencionan en el PATH, lo cual es muy incómodo. Por eso decidí escribir el comando ADONDE.bat que encuentra un programa en el PATH actual. En la Figura 3 está el resultado de ejecutar ADONDE.bat para encontrar los nombres de archivos cuyo prefijo es "comm", y luego "adonde".

ADONDE.bat recibe dos argumentos. El primero es un prefijo del nombre del comando que uno busca, y el segundo es una lista extra de directorios en dónde buscar, además de los directorios mencionados en la variable de ambiente PATH. ADONDE.bat imprime todos los archivos que corresponden al comando buscado.

Mi primera idea para implementar ADONDE.bat fue crear el comando que está en la Figura 4 (los puntos suspensivos indican que el comando completo debe escribirse en una sola línea).

```
for %%d in (.;%path%;%2;%3;%4) do
if exist %%d\%1*.* for %%f
in(%%d\%1*.*) do echo %%f
```

La explicación del comando de la Figura 4 es la siguiente: el primer FOR sirve para sacar de la variable de ambiente PATH cada uno de los directorios. Para que la búsqueda incluya al directorio actual se agrega ".;" al principio, y al agregar a los argumento %2;%3;%4 al final, entonces la búsqueda también incluirá a los directorios definidos por esos argumentos.

```
:: ADONDE.bat ==> Encuentra un archivo en el PATH actual
for %%d in (.;%path%;%2;%3;%4) do ...
 ... if exist %%d\%1*.* for %%f in (%%d\%1*.*) do echo %%f
:: ADONDE.bat ==> Fin de archivo
```

Figura 4

En la Figura 5 está el valor que recibe en cada argumento ADONDE.bat cuando se le invoca.

El IF en la Figura 4 sirve para determinar si existe, en alguno de los directorios escogidos por el primer FOR, algún archivo de los denotados por la máscara que ADONDE.bat recibe como primer argumento (%1). Si ese archivo existe el segundo FOR se encarga de desplegarlo en pantalla. Para no obligar al usuario de ADONDE.bat a escribir el nombre completo del comando que busca, en el IF se agrega un asterisco al nombre del archivo buscado. Por ejemplo, si el valor de "%%d" es "C:\DOS", y si "%1" es "comm*", entonces el IF evaluará a VERDADERO si existe el archivo "C:\DOS\comm**.*" (con tres asteriscos). Esta máscara denota a los dos archivos

```
:: ADONDE.bat ==> Encuentra un archivo
en el PATH actual
@echo off
@if not (%_depure%)==()
echo %0 %1 %2 %3 %4
@for %%d in (.;%path%;%2;%3;%4)
do call _%0 +++ %%d %1
:: ADONDE.bat ==> Fin de archivo

:: _ADONDE.bat ==> FOR anidado para
ADONDE.bat
@echo off
@if not (%_depure%)==()
echo %0 %1 %2 %3
@if exist %2\%3*.*
for %%f in (%2\%3*.*) do echo %%f
:: _ADONDE.bat ==> Fin de archivo
```

Figura 6

```
C:\ARTICULO>adonde comm* c:\bin;e:\tmp  a:
                %0     %1    %2     %3      %4

(.;%path%;%4)==(.;C:\UTL;C:\BAT;C:\DOS;a:)
```

Figura 5

C:\DOS\COMMAND.620 y C:\DOS\COMMAND.COM que aparecen en la Figura 3; el FOR anidado en la Figura 4 sirve para obtener el nombre exacto del archivo (por ejemplo, C:\DOS\COMMAND.620), a partir de la máscara (C:\DOS\comm**.*). La labor del FOR anidado es precisamente desplegar el nombre completo del archivo. De hecho, sin este FOR, ADONDE.bat sólo desplegaría la máscara (C:\DOS\comm**.*).

A pesar del análisis teórico de los últimos párrafos, al correr esta primera versión de ADONDE.bat la computadora desplegó el mensaje de error "FOR no se puede anidar". No queda más que partir el comando ADONDE.bat en dos, para que no queden los comandos FOR anidados. Para eso hay que crear un comando nuevo en el archivo _ADONDE.bat (con guión al principio), e invocarlo desde ADONDE.bat (sin guión al principio) mediante el comando "CALL" de forma que, al terminar la invocación de _ADONDE.bat, el control de ejecución vuelva al comando llamador ADONDE.bat.

El resultado de esta cirugía se muestra en la Figura 6. El nombre con que ha sido invocado un comando DOS siempre es el argumento número cero del comando (%0). Por eso en la invocación:

```
call _%0 +++ %%d %1
```

el argumento %0 es sustituido por la hilera "ADONDE". Al concatenarle al guión (_) esta hilera, resulta el nombre "_ADONDE" del comando auxiliar que se debe invocar.

Con esta división en dos archivos de ADONDE.bat se obtienen los resultados que aparecen en la Figura 3. La letra "@" (símbolo de arroba) al principio de cada comando evita que DOS despliegue en la pantalla cada comando antes de procesarlo; la variable de ambiente "_depure" sirve para desplegar los valores de los argumentos con que los comandos ADONDE.bat y _ADONDE.bat son invocados. El resultado de usar esta traza de ejecución, algo rudimentaria, se muestra en la Figura 7. El nombre del comando que se invoca es lo que aparece primero en cada renglón (%0).

```
C:\ARTICULO>set
path=C:\UTL;C:\BAT;C:\DOS
C:\ARTICULO>set _depure=cualquier_cosa
C:\ARTICULO>adonde adonde C:\bin
e:\tmp;d:
adonde adonde C:\bin e:\tmp d:
_adonde +++ . adonde
_adonde +++ C:\UTL adonde
_adonde +++ C:\BAT adonde
C:\BAT\ADONDE.BAT
_adonde +++ C:\DOS adonde
_adonde +++ C:\bin adonde
_adonde +++ e:\tmp adonde
_adonde +++ d: adonde
```

Figura 7

La implementación de la Figura 6 es correcta pero tiene un problema muy grande: requiere de dos archivos de comandos, y eso es poco elegante. Lo lógico es fusionar estos dos archivos en uno solo, que contenga tanto el código de ADONDE.bat como el de _ADONDE.bat. Para lograrlo es necesario determinar cuál de las dos partes del código hay que ejecutar. Para esto se usa el primer argumento del comando: si es la hilera "+++", la sección de código a ejecutar es la que corresponde a _ADONDE.bat (con guión); de lo contrario se debe ejecutar la de ADONDE.bat (sin guión).

```
:: ADONDE.bat ==> Encuentra un
archivo en el PATH actual
::                - Versión consolidada
@echo off
@if not (%_depure%)==()
echo %0 %1 %2 %3 %4
@if (%1)==(+++) goto _adonde

:adonde
:: ADONDE.bat ==> Encuentra un
archivo en el PATH actual
@for %%d in (.;%path%;%2;%3;%4)
 do call %0 +++ %%d %1
goto _fin

:_adonde
:: _ADONDE.bat ==> FOR anidado para
ADONDE.bat
@if exist %2\%3*.* for %%f in
(%2\%3*.*) do echo %%f
:: Fin de archivo: _ADONDE.bat

:_fin
:: ADONDE.bat ==> Fin de archivo
```

Figura 8

La Figura 8 es el resultado de consolidar, en un solo archivo, los dos archivos de la Figura 6. La hilera "+++" fue elegida porque ningún programa se puede llamar "+++.exe" (DOS no permite que el nombre de un archivo incluya el carácter "+"); esta hilera se usa para separar el código de _ADONDE.bat del de ADONDE.bat, y el primer IF permite distinguirlos.

```
C:\ARTICULO>set
path=C:\UTL;C:\BAT;C:\DOS
C:\ARTICULO>set _depure=cualquier_cosa
C:\ARTICULO>adonde adonde C:\bin
e:\tmp;d:
adonde adonde C:\bin e:\tmp d:
adonde +++ . adonde
adonde +++ C:\UTL adonde
adonde +++ C:\BAT adonde
C:\BAT\ADONDE.BAT
adonde +++ C:\DOS adonde
adonde +++ C:\bin adonde
adonde +++ e:\tmp adonde
adonde +++ d: adonde
```

Figura 9

Al correr la versión de ADONDE.bat de la Figura 8 con los mismos argumentos de la Figura 7, la traza de ejecución que se obtiene es la de la Figura 9. Estas dos trazas son prácticamente iguales, pues sólo difieren en el nombre del comando anidado ("adonde" vs "_adonde"). Como las dos trazas son tan similares, es natural pensar que el código de la Figura 6 es equivalente al de la Figura 8.

En la Figura 8 se usó el argumento "%0" para disfrazar el llamado que "ADONDE.bat" hace cuando invoca a "ADONDE.bat +++"; de hecho, la traza de ejecución de la Figura 9 no cambia si se remplaza "call %0" por "call adonde". En la Figura 10 se muestra una versión simplificada del ADONDE.bat de la Figura 8 en donde sí aparece el llamado (recursivo) que el comando ADONDE.bat se hace a sí mismo. El examen de este código brinda la primera luz sobre qué es y cómo funciona la recursividad.

los argumentos que recibe sólo cuando la variable de ambiente "_depure" está definida).

```
:: NUNCA.bat ==> Comando recursivo
que nunca termina...
@if not (%_depure%)==() echo %0 %1 %2
%3
@call NUNCA.bat %1 %2 %3
:: NUNCA.BAT ==> Fin de archivo
```

Figura 12

El examen del código del comando ADONDE.bat nos sirve para entender una de las aplicaciones de la recursividad: fundir en una sola dos o más rutinas relacionadas. En este ejemplo se nota que cuando se usa recursividad es muy importante que al ejecutar la rutina se retenga la posición desde donde se hace cada invocación; de la traza de la Figura 11 se puede ver la importancia de que, después de terminar la ejecución de cada una de las invocaciones recursivas (en _ADONDE.bat), se retorne al lugar desde donde se hizo el llamado (en el FOR que está en ADONDE.bat), pues de lo contrario no se ejecutarían los otros llamados recursivos.

```
C:\ARTICULO>set
path=C:\UTL;C:\BAT;C:\DOS
C:\ARTICULO>set _depure=cualquier_cosa
C:\ARTICULO>adonde mov d:
adonde mov
adonde +++    mov
```

Figura 13

La Figura 13 es la traza que muestra el efecto que tiene olvidar la posición de retorno en la invocación recursiva. Para obtener esta traza bastó eliminar del comando ADONDE.bat en la Figura 10 el comando "call". De esta forma DOS no recuerda la posición de retorno del comando. En esa traza hay sólo dos llamados: el inicial y uno recursivo. Como el recursivo no regresa al punto de invocación, el comando termina cuando ha procesado el argumento del directorio actual (.).

La aplicación de la recursividad que se ilustra con el comando ADONDE.bat no es la más común, pues cuando se usa recursividad no sólo es necesario recordar la dirección de retorno para cada llamado recursivo, sino que también es necesario usar memoria adicional para almacenar el valor de las variables locales usadas en cada invocación. En la siguiente sección se ilustra esto usando la función factorial.

2. FACT(): UNA FUNCIÓN PASCAL RECURSIVA

Un ejemplo que con frecuencia se usa para explicar la recursividad es la función entera factorial, que se denota n!. El factorial de "n" se calcula multiplicando todos los números desde "1" hasta "n".

```
Fact(n)   := n * (n-1 * ... * 2 * 1)   :=  n!
Fact(n-1) :=     (n-1 * ... * 2 * 1)   := (n-1)!


Fact(n)   := n * Fact(n-1)             :=  n!
```

Figura 14

De acuerdo a la definición de n!, para cualquier número entero se dan las igualdades de la Figura 14, en donde el factorial se define en términos de sí mismo: esta es la esencia de la recursividad.

Al desarrollar la fórmula del factorial de la Figura 14 para el caso específico n = 4 se obtiene el desglose que se muestra en la Figura 15.

```
1! := Fact(1) := 1;
   := 1;2! := Fact(2) := 2;
   := 2 * 1;
   := 2 * Fact(1);3! := Fact(3) := 6;
   := 3 * 2 * 1;
   := 3 * Fact(2);4! := Fact(4) := 24;
   := 4 * 3 * 2 * 1;
   := 4 * Fact(3);
```

Figura 15

Si se desglosa el factorial del número 3 usando la fórmula de la Figura 14, se obtiene la secuencia de cálculos que están en la Figura 16 en donde se ilustra cómo se puede calcular el valor de 3! = Fact(3) usando boletas para apuntar los resultados intermedios [Sth-92].

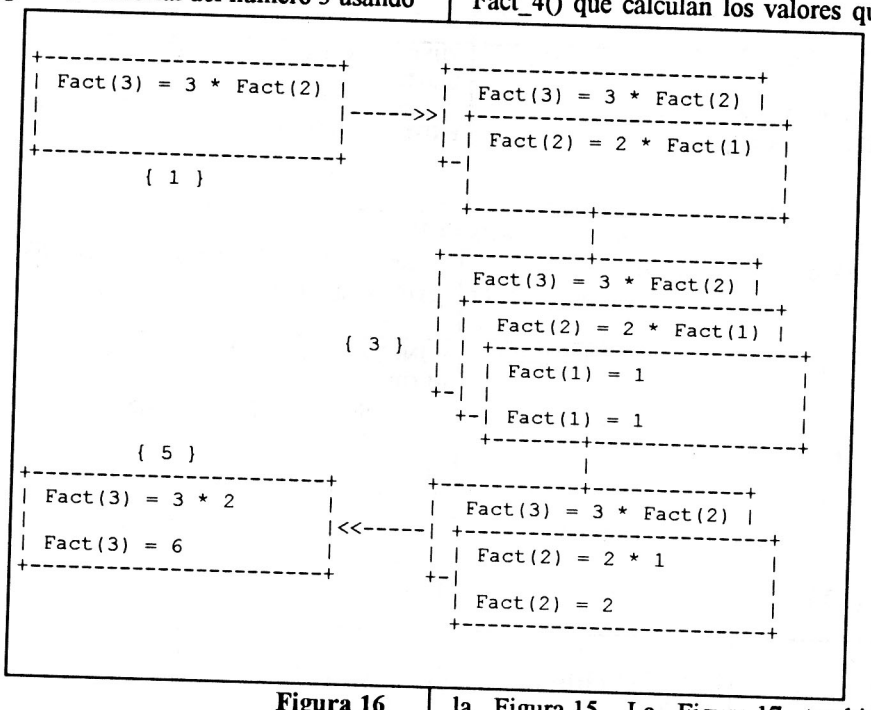


Figura 16

Conforme se avanza en el desarrollo de las fórmulas es necesario usar una nueva boleta para registrar los resultados intermedios; cuando por fin se llega al cálculo del factorial de 1, se pueden desechar las boletas más recientes, hasta que al final se obtiene el resultado deseado

La Figura 17 es un resumen de los cálculos que aparecen en las boletas de la Figura 16; cada columna corresponde a un cálculo intermedio en las boletas.

La Figura 18 es la implementación de las cuatro funciones Fact_1(), Fact_2(), Fact_3() y Fact_4() que calculan los valores que aparecen la Figura 15. La Figura 17 también puede interpretarse como la traza de ejecución de un programa como el de la Figura 18.

El siguiente paso para escribir una función que calcule el factorial es notar que las funciones Fact_2(), Fact_3() y Fact_4() de la Figura 18 son muy similares, pues lo que hacen es invocar a la función que calcula el factorial inmediatamente inferior. Sin embargo, la función Fact_1() es diferente pues no invoca a otra función, y siempre retorna el mismo valor: 1. En otras palabras, para implementar la función Fact_3() que calcula 3! basta copiar el código de la función Fact_4(), y sustituir el 3 por un 2, y el 4 por un 3. O sea, que la implementación de las función Fact_4() se puede usar como plantilla para obtener la implementación de la función

{1}	{2}	{3}	{4}	{5}
3*Fact(2)	3*Fact(2) 2*Fact(1)	3*Fact(2) 2*Fact(1) 2*1	3*2 2	6

Figura 17

```
FUNCTION Fact_1 : INTEGER;
BEGIN
  Fact_1 := 1;
END; { Fact_1 }
```

```
FUNCTION Fact_2 : INTEGER;
BEGIN
  Fact_2 := 2 * Fact_1;
END; { Fact_2 }
```

```
FUNCTION Fact_3 : INTEGER;
BEGIN
  Fact_3 := 3 * Fact_2;
END; { Fact_3 }
```

```
FUNCTION Fact_4 : INTEGER;
BEGIN
  Fact_4 := 4 * Fact_3;
END; { Fact_4 }
```

Figura 18

general Fact_n(), para cualquier n!.

Cabe ahora analizar las funciones F3(), G2() y H1() de la Figura 19.

```
FUNCTION F3 (n3 : INTEGER) : INTEGER
BEGIN
        IF n3 <= 1 THEN BEGIN
        F3 := 1
END;
ELSE BEGIN
        F3 := n3 * G2(n3 -1); {2}
 END;
END; { F3 }
```

```
FUNCTION G2(n2 : INTEGER) : INTEGER
BEGIN
        IF n2 <= 1 THEN BEGIN
        G2 := 1
END;
ELSE BEGIN
        G2 := n2 * H1(n2-1); { 3}
END;
END; { G2 }
```

```
FUNCTION H1(n1 : INTEGER) : INTEGER;
BEGIN
        IF n1 <= 1 THEN
        H1 := 1;            {4}
END
ELSE BEGIN
        H1 := n1 * Q0(n1-1);
 END;
END; { H1 }
```

Figura 19

Invocación	{1}	{2}	{3}
a := F3(3) G2(2) H1(1)	{1} @F3(3) n3=3	{1} @F3(3) n3=3 {2} @G2(2) n2=2	{1} @F3(3) n3=3 {2} @G2(2) n2=2 {3} @H1(1) n1=1

Figura 20

En la Figura 20 está la traza de ejecución que se obtiene al ejecutar la instrucción a := F3(3). Cada vez que se invoca un procedimiento es necesario crear un registro de activación [Pra-87], que contiene dos cosas: la dirección de retorno de la rutina, y espacio para las variables locales que usa. En la Figura 20 se denota la dirección de retorno precediendo el nombre de la función con el símbolo "@"; a la par aparece la variable local que genera cada invocación.

Por ejemplo, cuando desde la función F3() se invoca a G2(), que se muestra con la anotación {2} en la Figura 19, se crea el registro de activación [@G2(2) | n2=2], que aparece bajo la columna {2} en la Figura 20.
El registro de activación [@G2(2) | n2=2] indica que al terminar de ejecutar la función G2() se debe regresar al punto marcado {2} en la Figura 19 (la invocación inicial {1} está en el programa principal: a := F3(3)). El valor n2 = 2 indica que para ejecutar G2() es necesario crear una variable local llamada "n2" cuyo valor es "2". No es el caso para estas funciones, pero en general las rutinas usan más de una variable local, en cuyo caso el registro de activación tendría más de una variable:

[@Rutina(2,3) | a=2,b=3,c,d,e]

En este ejemplo, Rutina(a,b) tiene dos parámetros llamados (a,b) y usa tres variables locales (c,d,e).
Usar registros de activación permite hacer cualquier cantidad de invocaciones entre procedimientos, sujetos sólo a la cantidad de memoria disponible para crear nuevos registros de activación. Como se muestra en la Figura 21, al terminar cada procedimiento los registros de activación son removidos en orden inverso a como fueron creados. La estructura de datos que se ajusta a este comportamiento es la pila, por lo que el conjunto de registros de activación se conoce como la pila de ejecución del programa.

	Retorna { 1 }	Retorna { 3 }	Retorna { 2 }
a := F3(3) G2(2) H1(1)	@F3(3) n3=3 @G2(2) n2=2 @H1(1) -> 1	@F3(3) n3=3 @G2(2) -> 2	@F3(3) -> 6

Figura 21

Los primeros lenguajes de programación, como Fortran y Cobol, no usaban una pila de registros de activación para ejecutar los programas, lo que explica por qué en esos lenguajes no se podía usar recursividad. Todos los lenguajes modernos usan una pila de variables locales por dos razones: es muy sencillo crear cada contexto de ejecución para una rutina, y automáticamente cualquier procedimiento puede ser recursivo.

De forma natural surge la recursividad si se examina con cuidado el código de las funciones F3(), G2() y H1(). Salvo el cambio de nombre, todas estas funciones son exactamente equivalentes, pues el algoritmo que cada una de ellas implementa es exactamente el mismo: llamar a la función que calcula el factorial inmediatamente inferior o retornar el valor 1 si el argumento de entrada es 1. De hecho, esta descripción es la misma de las funciones Fact_1()...Fact_4() de la Figura 18. La diferencia está en que F3(), G2() y H1() tienen un parámetro (n3, n2 o n1) que se usa para hacer el cálculo.

Más aún, si hacemos un examen más detallado de estas tres funciones podemos encontrar, como es el caso del comando ADONDE.bat de la Figura 10, los dos componentes que siempre están en un procedimiento recursivo. Primero, está la verificación del caso límite, que en este caso es el IF que prueba la condición (n<=1), y el otro es la invocación recursiva del procedimiento, que en este caso se da cuando F3() invoca a G2(), G2() a H1(), y H1() a Q0().

¿Cómo sería la versión recursiva de las funciones F3(), G2() y H1()? En la Figura 22 se muestra la función Fact() recursiva que se ha

```
FUNCTION Fact(n : INTEGER) : INTEGER;
{ RESULTADO
  Calcula el factorial (n!) del número
"n".
  - Esta es la implementación recursiva
clásica. }
BEGIN
  IF n <= 1 THEN BEGIN
    Fact := 1;
  END
  ELSE BEGIN
    Fact := n * Fact(n-1);     { * }
 END;
END;  { Fact }
```

Figura 22

obtenido editando la implementación de F3() y haciendo las siguientes sustituciones con el editor de textos:

[F3==>Fact G2==>Fact n3==>n].

La Figura 23 es la traza de ejecución de la invocación a := Fact(3), y es muy similar a la de la Figura 20. Las diferencias que estas dos trazas presentan son las siguientes:

Invocación	{1}	{2}	{3}
a := F3(3) 3*Fact(2) 2*Fact(1)	@Fact(3) n=3	@Fact(3) n=3 @{*} n=2	@Fact(3) n=3 @{*} n=2 @{*} n=1

Figura 23

1. Para evitar confusión, en la Figura 20 se usaron como nombre del parámetro para cada función F3(), G2() y H1() los identificadores n3, n2 y n1. En realidad se pudo haber usado el mismo nombre "n" en todos los casos sin problema alguno, pero tal vez eso hubiera opacado el hecho de que cuando cada una de esas funciones es invocada, necesita memoria adicional para almacenar sus variables locales. Esta memoria se obtiene de la pila de ejecución del programa.

En la Figura 23 se tiene que usar el mismo nombre para el parámetro "n", pero en cada invocación recursiva de la función Fact() es necesario separar memoria de la pila de ejecución del programa, por lo que, aunque en la Figura 23 aparece el mismo nombre "n" varias veces, en realidad cada una de las invocaciones está usando una variable diferente.

2. En la Figura 20 la dirección de retorno de F3(3) es el programa que hace el llamado inicial {1}, que también es la dirección de retorno de la invocación inicial a Fact(3). Sin embargo, cada una de las invocaciones recursivas de Fact() debe regresar al punto marcado con {*} en la Figura 22, que es el punto equivalente de retorno desde H1() a G2(), y desde G2() a F3() en la Figura 19. Esos son los puntos de retorno recursivos.

Si se examina el código de Fact() es fácil detectar que el llamado recursivo se hace con un argumento diferente al recibido por la función, pues si entra el valor "n", la invocación recursiva se hace con "n-1". En general, cuando se escriben algoritmos usando recursividad, lo usual es que sea muy obvia cuál es la modificación que hay que hacerles a los argumentos que recibe el procedimiento antes de hacer el llamado recursivo. De esta forma se evita que haya ciclos recursivos infinitos, como el que se ilustra en el comando NUNCA.bat de la Figura 12.

En resumen, la recursividad se basa en el hecho de que cuando un procedimiento es invocado, obtiene una copia nueva de todas sus variables locales, que generalmente están en la pila de ejecución del programa; ahí es donde queda también registrado el lugar de retorno para la rutina. Cuando el procedimiento recursivo trabaja sobre sus variables, está usando las que están en el nuevo registro de activación, por lo que, cuando termina su trabajo, regresa al punto de invocación que le corresponde. En realidad la recursividad existe porque el llamado entre procedimientos se implementa usando registros de activación organizados en una pila de ejecución para el programa.

En la implementación de Fact() están las dos partes que siempre aparecen en un procedimiento recursivo, separadas por la instrucción IF. La prueba (n<=1) es la que verifica si ya se cumple la condición para parar los llamados recursivos; en el otro bloque del IF aparece la invocación recursiva.

3. PID (R) : RECORRIDO RECURSIVO DE UN ÁRBOL

Con frecuencia se usa el árbol como ejemplo para que el estudiante aprenda recursividad,

pues el árbol es la estructura de datos recursiva por excelencia. Un árbol binario se define, recursivamente, como sigue:
El árbol vacío es un árbol binario.
Si T no está vacío, entonces T contiene un nodo raíz que a su vez tiene enlazados a la izquierda un árbol binario, y a la derecha otro.

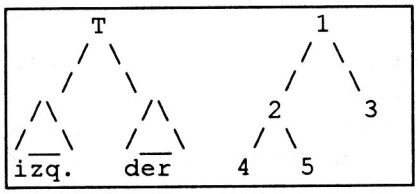


Figura 24

En la Figura 24 se muestran el diagrama general de un árbol binario, y el árbol [1 [2 [4 5]] 3].

```
TYPE
  PNodo = ^TNodo;
  TNodo =  RECORD
     izq,                { puntero al hijo
izquierdo }
     der : PNodo;        { puntero al hijo
derecho    }
     info: CHAR;         { contenido del
nodo          }
   END; { TNodo }
```

Figura 25

Para implementar un árbol en Pascal lo usual es usar nodos como los que se definen en la Figura 25. Un árbol entonces es un puntero a su nodo raíz, que puede ser NIL si el árbol está vacío. Si un nodo no tiene alguno de sus hijos, entonces el puntero correspondiente es NIL.

Como los árboles tienen muchos usos, es necesario contar con algoritmos para recorrerlos y procesar su contenido. Una forma muy usada para procesar árboles es el PreOrden, o recorrido PID, conocido así por las siglas de las palabras Procese-Izquierda-Derecha que definen el orden de acceso a los nodos del árbol.
Para recorrer un árbol en PID lo que se hace es procesar primer la raíz (por eso Procese aparece primero), y luego se procesa (recursivamente) primero el hijo izquierdo del nodo, y luego el derecho. El orden de recorrido PID para el árbol binario de la Figura 24 es [1 2 4 5 3]. En la siguiente traza se muestra entre paréntesis cuadrados [] el nodo visitado.

`[1]` Para comenzar, se *Procesa* la raíz del árbol.

`[2]` Luego del paso anterior se procesa al hijo *Izquierdo* de la raíz. Para este subárbol se aplica de nuevo el orden de recorrido PID por lo que se *Procesa* la raíz de este subárbol `[2]`.

`[4]` El subárbol *Izquierdo* del nodo `[2]` es un árbol unitario que sólo contiene al nodo [4].

`[]` Se aplica el proceso PID al subárbol *Izquierdo* del nodo `[4]`, pero como ese subárbol está vacío, no se obtiene valor alguno.

`[5]` Se aplica el proceso PID al subárbol *Derecho* del nodo `[2]`, que es un árbol unitario que sólo contiene al nodo `[5]`.

`[]` Se aplica el proceso PID al subárbol *Izquierdo* del nodo `[5]`, pero como ese subárbol está vacío, no se obtiene valor alguno. Se ha terminado de recorrer el subárbol que tiene por raíz al nodo [2].

`[3]` Después de recorrer el hijo *Izquierdo* del nodo `[1]` se recorre al hijo *Derecho*, que es un árbol unitario que sólo contiene al nodo `[3]`. Se ha terminado de recorrer el

```
PROCEDURE PID(raiz: PNodo);
{ RESULTADO
  Imprime en PreOrden el contenido del
árbol
  cuya raíz es el nodo "raiz".
}
BEGIN
  IF raiz = NIL THEN BEGIN
    EXIT;                    { terminó la
recursión }
  END
  ELSE BEGIN
    Write(raiz^.info:2);   { P } {
Procesa    }
    PID(raiz^.izq);        { I } {
Izquierda }
    PID(raiz^.der);        { D } {
Derecha    }
  END;
END;  { PID }
```

Figura 26

subárbol *Derecho* de la raíz del árbol, por lo que el recorrido termina.

En la Figura 26 está el procedimiento recursivo PID() para recorrer árboles binarios. Como sucede en el caso de Fact() y de ADONDE.bat, el procedimiento recursivo PID() tiene dos partes. A diferencia de esos otros dos, PID() hace dos llamados recursivos: uno para el hijo izquierdo, y otro para el derecho.

Pero lo más importante de PID() es que muestra cómo la recursividad aumenta la expresividad del lenguaje, pues implementar el recorrido PreOrden para árboles sin usar recursividad es realmente complicado, pues sería necesario recordar (usando una pila) el nodo del subárbol que se está recorriendo. Más aún, para obtener otro tipo de recorrido del árbol, como IDP o DPI, basta cambiarles el orden a las instrucciones marcadas {P}, {I} y {D} en el procedimiento PID().

```
PROCEDURE EnPile;
{ RESULTADO
  C:\TMP>RevCHAR
  Digite la hilera a invertir:
  adolfo.
  .ofloda                          }
VAR
  ch : CHAR;
BEGIN { EnPile }
  Read(ch);

  IF ch  '.' THEN BEGIN
    EnPile;
  END;
  Write(ch);
END;  { EnPile }

PROCEDURE EnPileCH(VAR ch : CHAR);
{ RESULTADO
  C:\TMP>RevCHAR
  Digite la hilera a invertir:
  adolfo.  .......                 }
BEGIN { EnPileCH }
 ch := Crt.ReadKey;
 Write(ch);
 IF ch  '.' THEN
    BEGIN    EnPileCH(ch);
    END;  Write(ch);
END;  { EnPileCH }
```

Figura 27

Esta implementación de PID(), tan concisa y simple, pero sobre todo tan elegante, es la razón por la que todos los profesionales de la computación estamos obligados a dominar la recursividad. Esta técnica de programación permite implementar algoritmos muy elegantes.

4. OTROS ENFOQUES

La explicación de recursividad de [CC-85] es un buen complemento a las de este artículo, pues usa un procedimiento que invierte el orden de letras que lee recursivamente. Si el profesor imparte lecciones usando una pantalla gigante que despliega lo que ocurre en su computador, el ejecutar ante sus alumnos estos procedimientos le facilitará mcho explicar cómo funciona la recursividad.

En la Figura 27 está la implementación, tomada de [CC-85], de la rutina recursiva EnPile() que invierte el orden de una hilera. La condición de parada es la lectura del caracter ".", y es hasta ese momento que todos los llamados recursivos retornan. EnPile() despliega en orden inverso los caracteres que ha leído.

El programa RevCHAR.pas de la Figura 28 invoca a los procedimentos de la Figura 27. El procedimiento EnPileCH() de la Figura 27 sirve para ilustrar por qué es importante que cada invocación recursiva use una copia diferente de las variables locales. Como el parámetro "ch" pasa de una invocación a la siguiente por referencia, todas las invocaciones comparten la misma variable "ch". Así EnPileCH() siempre imprime solamente la última letra que recibió, la que forzosamente debe ser ".". A los estudiantes esto los impresiona mucho, más cuando examinan con el depurador simbólico los valores almacenados en la pila de ejecución del programa.

```
PROGRAM RevCHAR;
{ RESULTADO
  Vuelve al revés los caracteres
leídos. }

PROCEDURE EnPile;                    {
... }
PROCEDURE EnPileCH(VAR ch : CHAR);   {
... }

VAR
  ignore: CHAR;
BEGIN { RevCHAR }
  WriteLn;
  WriteLn('Digite la hilera a invertir:
');
  ignore := '^';

  EnPileCH(ignore);  WriteLn;   { 1 }
  EnPile;  ReadLn;     WriteLn;
{ 2 }
END.   { RevCHAR }
```

Figura 28

Es importante conocer los principios teóricos de la recursividad. En [ASU-86] los autores explican con gran profundidad qué es un registro de activación. Una exposición más simple de estos conceptos está en [Pra-87]. En [Bog-96] están los fundamentos matemáticos de la recursividad, en un enfoque diseñado para estudiantes de computación.

El enfoque de [Gld-93] es el de la mayoría de los textos de programación. La idea de usar las boletas de la Figura 16 fue tomada de [Sth-92], pp 44.

```
PROCEDURE BajeYSuba( primero, n :
INTEGER );
{ Como ejercicio, determine qué
despliega
  este programa, y porqué lo hace.
}
BEGIN
  IF n > 0 THEN BEGIN
    INC(primero); DEC(n);
    WriteLn( 'Pre[', primero:1,',',
n:1,']' );
    BajeYSuba( primero, n );
    WriteLn( 'Pos[', primero:1,',',
n:1,']' );
  END;
END;
```

Figura 29

5. CONCLUSIÓN

La recursividad es una técnica de programación mediante la cual un módulo puede invocarse a sí mismo. Es importante porque algunos algoritmos recursivos son mucho más compactos y elegantes que los algoritmos no recursivos equivalentes. Por eso quien domina las técnicas básicas de programación debe saber usar recursividad.

Cuesta bastante entender por primera vez qué es recursividad, pero para hacerlo ayuda mucho conocer cómo se usan los registros de activación en la pila de ejecución del programa, para recordar la dirección de retorno del procedimiento y almacenar sus variables locales.

6. RECONOCIMIENTOS

Esta investigación se realizó dentro del proyecto de investigación 326-93-256 "DBgen: generación automática de programas a partir de su base de datos", inscrito ante la Vicerrectoría de Investigación de la Universidad de Costa Rica. La Escuela de Ciencias de la Computación e Informática también ha aportado fondos para este trabajo.

7. BIBLIOGRAFÍA

[Arg-91] Argüello, José Rónald: *Estándares para sistemas de archivos en computadores*, Revista de la Facultad de Ingeniería, Universidad de Costa Rica, Vol 1 No. 1, pp [85-92], Marzo 1991.

[ASU-86] Aho, Alfred V.; Sethi, Ravi; Ullman, Jefrrey D.: Compilers: Principles, Techniques, and Tools, Addisson Wesley Publishing Co., pp [401-409], 1986.

[Bog-96] Bogart, Kenneth P.: Matemáticas discretas, Editorial Limusa, S.A. de C.V., Grupo Noriega Editores, ISBN 968-18-5218-4, México, 1996.

[CC-85] Cooper, Doug; Clancy, Michael: Oh! Pascal!, 2nd edition, W.W. Norton & Company, pp [234-246], New York and London, 1985.

[Gld-93] Goldstein, Larry Joel: Turbo Pascal: Introducción a la Programación Orientada a Objetos, Prentice-Hall Hispanoamericana, ISBN 0-13-629635-1, pp [190-198], 1993.

[Pra-87] Pratt, Terrence W.: Lenguajes de Programación: Diseño e implementación, 2da edición, Prentice-Hall Hispanoamericana, ISBN 0-13-730580-X, pp [226-292, 293-317], 1997.

[Sth-92] Sethi, Ravi: Lenguajes de Programación: conceptos y constructores, Addisson-Wesley Iberoamericana, ISBN 0-201-51858-9, pp [41-46, 128-136, 142-144], 1992.