Subprogramas. Funciones y Procedimientos

Programación 1

InCo - FING

Subprogramas

Ejemplo

```
program triangulo;
var base_triangulo, altura_triangulo, area_triangulo : real;
    (* declaracion de subprogramas
LeerDatos, CalcularAreaTriangulo, MostrarResultado
    *)
procedure LeerDatos(var base,altura : real);
 begin
  ...
 end; { LeerDatos }
function CalcularAreaTriangulo(base,altura : real) : real;
 begin
  ...
 end; { CalcularAreaTriangulo }
procedure MostrarResultado(area : real);
 begin
  ...
 end; { MostrarResultado }
begin (* programa principal *)
   LeerDatos(base_triangulo,altura_triangulo);
   area_triangulo:=
      CalcularAreaTriangulo(base_triangulo,altura_triangulo);
   MostrarResultado(area_triangulo);
end.
```

Introducción

Un **subprograma** es un fragmento de código que se comporta de manera independiente dentro de un programa.

Los subprogramas pueden ser invocados varias veces desde otras partes del programa.

Se comunican mediante el *pasaje de parámetros*.

Cada subprograma tiene su propio espacio de nombres (identificadores *locales*)

Algunos identificadores pueden ser compartidos entre subprogramas y el programa principal (identificadores *globales*).

Los subprogramas son una herramienta de *modularización*.

Estructura de un bloque

bloque es una denominación genérica para la siguiente estructura sintáctica.

```
const
   <declaraciones de constantes>

type
   <declaraciones de tipos>
var
   <declaraciones de variables>

<declaraciones de subprogramas>
begin
  <instrucciones>
end
```

Encabezados

Todo bloque viene precedido de un encabezado:

programa **program** *identificador*;

procedimientos **procedure** *nombre*(*parametros* . . .);

funciones **function** *nombre*(*parametros*. . .) : tipo;

Funciones

Introducción

- Una **función** es un subprograma que *retorna* un valor de tipo simple.
- Las funciones se invocan dentro de una *expresión*.
- Funciones estándar: `ord`, `succ`, `pred`, `sqrt`, `chr`, `trunc`, etc.
- Funciones definidas por el programador: se declaran en el programa luego de la declaración de variables.

Declaración de una función

Sintaxis:

```
function  nombre ( lista_de_parametros )  : tipo;
const
   ...
type
   ...
var
   ...
(* subprogramas *)
...
begin
  ...
end;
```

Encabezado de una función

function *identificador* (`parametros`) : *tipo*;

- `function` es una palabra reservada.
- *identificador* debe ser único.
- *tipo* debe ser un tipo **simple** (integer, char, boolean)[1].

[1] Existen tipos **estructurados** (arreglos, registros), que se estudian más adelante.

Parámetros

La lista de parámetros tiene esta forma:

```
listaparametros = parametros {';' parametros}

parametros
      = ['var'] identificador {',' identificador} ':' tipo
```

Nota: `tipo` **no** puede ser anónimo

Ejemplos:

```
a,b,c: integer; var a : arreglo; var error,salir: boolean

var a: real; c: char; var m: integer
```

Parámetros nominales y efectivos

Los **parámetros nominales** (también llamados **formales**) son los *nombres* que aparecen en el encabezado de la función:

```
(* base y exponente son parámetros nominales *)
function potencia(base: real; exponente: integer): real;
```

Los **parámetros efectivos** (también llamados **verdaderos**) son las *expresiones* que aparecen en la invocación de la función.

```
pot:= potencia(pi,23);
...
WriteLn(potencia(2*pi*sqr(radio),N+2));
```

Para cada parámetro, el tipo del parámetro nominal y el tipo del respectivo parámetro efectivo deben ser **compatibles**.

Ejemplo: la función **potencia**

```
function potencia(base : real; exponente : integer) : real;
(* pre condicion: (base<>0) or (exponente<>0) *)
var pot       : real;
    i         : integer;
    negativo : boolean;
begin
   negativo:= exponente < 0;
   exponente:= abs(exponente);
   pot:= 1;
   for i:= 1 to exponente do
      pot:= pot * base;
   if negativo then
       potencia:= 1 / pot
   else
       potencia:= pot
end;
```

Ejemplo de invocación

Calcular las potencias de un número elevado a 10 exponentes que son leídos de la entrada.

```
ReadLn(base);
for i:= 1 to 10 do
begin
  read(n);
  WriteLn(base:6:2,
          '^',
          n:2,
          '=',
          potencia(base,n):10:2)
end;
```

Nombre de la función

El nombre de la función es un identificador.
Se utiliza para especificar el valor que retorna la función.

```
function potencia ....

begin
  ...
  potencia:= valor; (* valor retornado *)
  ...
end
```

Observación: `potencia` **no** es una variable. No puede "utilizarse" su valor.

```
(* incorrecto *)
potencia:= potencia * base
```

Ejemplo. Funciones booleanas

Verificar si un número es primo:

```
function EsPrimo(numero: integer): boolean;
var i,tope: integer;

  function divide(n,m: integer): boolean;
  begin
    divide:= m mod n = 0;
  end;

begin
  i:= 2;
  tope:= trunc(sqrt(numero));
  while (i<=tope) and not divide(i,numero) do
      i:= i+1;
  EsPrimo:= i > tope
end;
```

Procedimientos

Introducción

- Los procedimientos no retornan un valor en su nombre.
- Se invocan como una instrucción independiente.
- El encabezado de un procemiento tiene esta forma:

```
procedure nombre ( parametros ... );
```

Ejemplo. Procedimiento de salida.

Mostrar el resultado de calcular el área de un triángulo.

```
procedure MostrarResultado(area : real);
   begin
      WriteLn;
      WriteLn( '                                    **********');
      WriteLn( '  El area del triangulo es:   ', area:8:2);
      WriteLn( '                                    **********');
      WriteLn;
   end; { MostrarResultado }
```

Pasaje de parámetros

Pasaje de parámetros por **valor**

Pasaje por valor: Son los parámetros **no** precedidos por `var`

- En el momento de la invocación se realiza una copia de los valores de los parametros efectivos a los parámetros nominales.
- Los parámetros efectivos pueden ser *expresiones*

Ejemplo: Sea el cabezal `function f(a,b:integer) : boolean`
La invocación: `f(23,N*2)` equivale a lo siguiente:

```
(* pasaje de parámetros *)
a:= 23;
b:= N*2;
(* código de la función *)
...
```

Pasaje de parámetros por **referencia**

Pasaje por referencia: Son los parámetros precedidos por `var` (también se los denomina **parámetros de variables**)

- Los parámetros efectivos deben ser **variables**.
- En el momento de la invocación el parámetro nominal comparte el mismo espacio de memoria que el parámetro efectivo. (alias de variables).
- Toda modificación del parámetro nominal se refleja en el parámetro efectivo.
- En cambio, cuando el pasaje es por valor el parámetro efectivo no sufre modificaciones.
- No recomendamos utilizar pasaje por referencia para *funciones*.

Pasaje de parámetros por referencia: ejemplos (1)

Ejemplo. Procedimiento de entrada. Leer datos para el cálculo del área de un triángulo.

```
procedure LeerDatos(var base, altura : real);
    begin
       (* lectura de la base *)
       Write('Ingrese base del triangulo: ');
       ReadLn(base);
     {  el numero ingresado para la base es correcto  }
       (* lectura de la altura *)
       Write('Ingrese altura del triangulo: ');
       ReadLn(altura);
     { el numero ingresado para la altura es correcto  }
    end; { LeerDatos }
```

Notar la utilización de **var** en los parámetros.

Pasaje de parámetros por referencia: ejemplos (2)

Intercambio de variables.

```
(* el tipo T es cualquiera *)
procedure intercambio(var a,b: T);
var
  aux: T;
begin
  aux:= b;
  b:= a;
  a:= aux;
end;
```

Notar la utilización de **var** en ambos parámetros.

Recomendaciones de estilo

Recomendaciones de estilo: funciones

- No utilizar pasaje por referencia con funciones.
- No hacer entrada y salida dentro de funciones (`read`, `write`, etc)
- No utilizar variables globales (declaradas en el programa principal) dentro de subprogramas.
- Asignar una sola vez y al final el valor de la función.
- Definir funciones para todo cálculo intermedio que sea independiente.
- Sólo definir funciones cuya semántica sea clara.

Recomendaciones de estilo: procedimientos

Ubique sus procedimientos en alguna de las siguientes clases:

- **Salida**. Despliegan resultados en la salida. Estos procedimientos no hacen entrada. No tienen parámetros por referencia.
- **Entrada**. Ingresan datos desde la entrada y lo cargan en variables. No hacen salida, salvo para "pedir" los datos. Sus parámetros son por referencia.
- **Internos**. NO hacen entrada-salida. Reciben datos del programa y retornan éstos modificados. Contienen los dos tipos de parámetros.
- NO utilice variables globales. Todos los valores compartidos deben pasarse como parámetros.

Ejemplo completo

```
program triangulo;
    var base_triangulo, altura_triangulo, area_triangulo : real;
    (* declaracion de subprogramas *)
    procedure LeerDatos(var base,altura : real);
    begin
       (* lectura de la base *)
     Write('Ingrese base del triangulo: ');
       ReadLn(base);
       while base <= 0 do  (* validacion *)
       begin
          WriteLn('La base debe ser un real positivo');
          Write('Ingrese base del triangulo: ');
          ReadLn(base);
       end;
     (* lectura de la altura *)
       Write('Ingrese altura del triangulo: ');
       ReadLn(altura);
       while altura <= 0 do   (* validacion *)
       begin
          WriteLn('La altura debe ser un real positivo');
          Write('Ingrese altura del triangulo: ');
        ReadLn(altura);
     end;
    end; { LeerDatos }
```

Ejemplo completo (continuación)

```
function CalcularAreaTriangulo(base,altura : real) : real;
 begin
    CalcularAreaTriangulo:= base * altura / 2;
 end; { CalcularAreaTriangulo }

procedure MostrarResultado(area : real);
 begin
  WriteLn;
    WriteLn( '                                  **********');
    WriteLn( '  El area del triangulo es:  ', area:8:2);
    WriteLn( '                                  **********');
    WriteLn;
 end; { MostrarResultado }

 begin (* programa principal *)

    LeerDatos(base_triangulo,altura_triangulo);

    area_triangulo:=
       CalcularAreaTriangulo(base_triangulo,altura_triangulo);

    MostrarResultado(area_triangulo);
 end.
```

Reglas de alcance

Alcance

- El **alcance** de un identificador es aquella porción del programa en que dicho identificador es visible.
- Existen reglas de alcance que definen la visibilidad de cada identificador.

Identificadores locales y globales

Un identificador definido en un bloque es visible en ese bloque y en todos los sub-bloques que contenga. No así en bloques externos.
Ejemplo:

```
procedure p(x,y: integer);
var z: integer;

    function f(a: integer) : integer;
    var b: integer;
    begin
      ... (* sentencias de f *)
    end;

begin
    ...   (* sentencias de p *)
end;
```

Identificadores locales y globales

- Los parámetros nominales `x` e `y` y la variable `z` son identificadores *locales* a `p` y *globales* a `f`.
- Es posible hacer referencia a `x`, `y` y `z` en las sentencias de `p` y dentro de `f`. **No** son visibles fuera de `p`.
- El parámetro nominal `a` y la variable `b` son *locales* a `f`. Se los puede referenciar únicamente en las sentencias de `f`.

Identificadores locales vs. globales

Los identificadores locales que tienen el mismo nombre que identificadores globales tienen prioridad sobre los globales. O sea, los locales "tapan" a globales de igual nombre.

```
procedure p(x,y: integer);
var z: integer;

    function f(x: integer) : integer;
    var b: integer;
    begin
      ... (* sentencias de f *)
    end;

begin
    ...   (* sentencias de p *)
end;
```

Identificadores locales vs. globales

- El parámetro nominal `x` de `p` es visible en las sentencias de `p`, pero no así dentro de `f`.
- Toda referencia a `x` dentro de `f` corresponde al parámetro nominal `x` de `f`, y **no** a la variable global `x`.

Funciones y procedimientos (1)

Los identificadores de funciones y procedimientos son visibles en el bloque donde están definidos y en todos los sub-bloques que siguen a su declaración (incluyendo el de su propia definición).

```
procedure p(x,y: integer);
var z: integer;

    function f(a: integer) : integer;
    var b: integer;
    begin
      ... (* sentencias de f *)
    end; {f}

    function g(c: real) : integer;
       procedure k(var d: real);
       begin
         ... (* sentencias de k *)
       end; {k}
    begin
      ... (* sentencias de g *)
    end; {g}

begin
    ...    (* sentencias de p *)
end; {p}
```

Funciones y procedimientos (2)

- La función `f` puede ser llamada:
  - desde las sentencias de `p` (por ser local a `p`).
  - dentro de las propias sentencias de `f` (llamada recursiva).
  - dentro de `g` (esto incluye al procedimiento `k`).
- La función `g` puede ser llamada:
  - desde las sentencias de `p`.
  - dentro de las propias sentencias de `g`.
  - desde las sentencias de `k` (por ser global a `k`).
- La función `g` **no** puede ser llamada desde `f`, por estar declarada después.
- El procedimiento `k` **no** puede ser llamado desde fuera de `g`, por ser local a `g`.