Universidad Complutense de
Madrid

Manualillo de estilo para programadores principiantes en Pascal

Informe técnico 109-00
Departamento de Sistemas Informáticos y Programación

C. Gregorio Rodríguez, L. F. Llana Díaz, R. Martínez Unanue,
P. Palao Gostanza, C. Pareja Flores y J. Á. Velázquez Iturbide

Manualillo de estilo
para programadores principiantes en Pascal

C. Gregorio, L. F. Llana, R. Martínez, P. Palao, C. Pareja y J. Á. Velázquez

Resumen

Entre las características de los programas que determinan su calidad, hay aspectos de fondo (corrección y eficiencia) y de forma (diseño, estilo y documentación). Cada vez más, se reconoce la importancia de estos últimos: repercuten directamente en los primeros, y también tienen interés por sí mismos, ya que el mantenimiento y reutilización de los programas depende básicamente de ellos. En estas páginas se recogen brevemente distintos aspectos relacionados con el estilo en programación imperativa en Pascal.

1 Introducción

El modelo imperativo es actualmente el más frecuentemente adoptado en cursos de introducción a la programación. Aunque el lenguaje Pascal no es desdeluego el único en estos cursos, ocupa un lugar insustituible hoy por hoy en un buen número de plantes de estudio universitarios.

Es corriente plantear muchas de estas asignaturas partiendo del objetivo general de la enseñanza de la programación:

> "Capacitar a los alumnos para construir metódicamente programas correctos, fáciles de mantener y reutilizar y eficientes"

Sin entrar ahora en muchos detalles sobre lo ambicioso que es este objetivo, sí podemos afirmar que la adquisición de unos buenos hábitos en el desarrollo de los programas incide directamente en la corrección, mantenimiento y reutilización. El proceso de desarrollo de los programas ha de ser metódico (diseño descendente y refinamientos sucesivos), y el programa final deberá responder a distintos criterios de calidad:

- La **corrección** es desde luego el objetivo principal de un programa: Un programa debe verificar su especificación. Por supuesto, no vamos a entrar aquí en este asunto. Pero sí señalamos que los aspectos que siguen (diseño, estilo y documentación) contribuyen al desarrollo de programas correctos, aunque se consideren por separado.

- El **diseño** atiende a distintos aspectos semánticos y metodológicos, como son la organización de los programas, el buen desarrollo de las distintas estructuras, subprogramas tipos de datos y TADs. La finalidad de un buen diseño es lograr programas que, además de correctos, sean mantenibles y reutilizables.

- El **estilo** y la **documentación** atienden a aspectos más formales como son el buen uso de los identificadores y comentarios, o la presentación del texto del programa. La finalidad es que los programas sean legibles con facilidad y naturalidad.

- La **eficiencia** es otro criterio de la calidad de los programas, pero en este folletito no lo tenemos en cuenta.

Ya se ve que el diseño, el estilo y la documentación no son los únicos objetivos importantes. E incluso su importancia puede considerarse secundaria en comparación con otros aspectos de la calidad de los programas. Pero, precisamente por ello, se ignora o desprecia con frecuencia el papel que desempeñan aspectos formales, como son la disposición, presentación, elección de identificadores, etc.

Ya Gries resaltó la importacia de estos aspectos [2], y subrayó su papel, no como adornos que decoran un programa desarrollado, sino como hábitos que todo programador debe practicar durante el proceso de la programación.

El objeto de estas páginas es justamente recoger estos aspectos.

2 Criterios generales

Un idea late bajo todo esto: que los programas han de ser ejecutados por máquinas, pero han de ser leídos y mantenidos por humanos. Es necesario por tanto facilitar la lectura de los programas y, por consiguiente, la comprensión de la lógica del programa, de qué hace y por qué. Para lograrlo, recogemos los siguientes objetivos para un programa, estrechamente relacionados entre sí:

- Programas **claros** y **naturales**. La finalidad es que un programa sea fácil de leer y entender, por distintos programadores, y no sólo por el que lo escribió. Como ideal, se puede plantear así: "que un programa se lea con la misma facilidad y naturalidad que si fuera una buena novela".
  La razón es facilitar el razonamiento sobre la corrección, su mantenimiento y su reutilización.

- Programas **autoexplicativos**: La idea básica es que mantener dos textos distintos para un mismo programa (el código y la documentación) propicia los errores, debido a la dificultad de mantenerlos consistentes [2].
  Por tanto, conviene integrar el programa y la explicación del mismo en un mismo documento. El gran énfasis que se pone en la documentación del programa recibe el nombre de *programación letrada* [3].

- Programas con un estilo **homogéneo** y **consistente**. Estas características contribuyen a facilitar la lectura de los programas.

Veremos seguidamente una serie de recomendaciones concretas con las que lograr programas que se adapten a los criterios anteriores.

3 Recomendaciones

3.1 Sobre las declaraciones

- La primera regla: usar identificadores significativos en el contexto del problema que se está resolviendo. Los identificadores impronunciables deben evitarse. No usar abreviaturas indiscriminadamente: sólo las consagradas (EEUU, pta, kg, etc.), que se entiendan por sí mismas.

- La única razón para limitar el tamaño de los identificadores es evitar el riesgo de colisiones entre nombres cuando se emplean compiladores que sólo reconocen una parte y truncan el resto.

- Para los datos constantes o variables, son apropiados los sustantivos, salvo con los de tipo booleano, que se bautizan bien mediante afirmaciones (que pueden ser ciertas o falsas):

```
CONST
   PorCiento          = 0.01;
VAR
   sueldo             : real;
   distancia          : real;
   inicial            : char;
   monedero           : integer;
   cuentaBanco        : integer;
   quedanFondosEnCaja: boolean;
```

  Obsérvese la naturalidad de su uso en el texto del programa:

```
sueldo:= sueldo * (1 + 14 * PorCiento);
cuentaBanco:= cuentaBanco + sueldo;     {cobro, a primeros de mes}
```

- Los procedimientos se pueden nombrar adecuadamente con identificadores que representan verbos en infinitivo:

```
SacarCantidadMetalico(cuentaBanco, monedero, 10000)
```

  También es adecuado usar el imperativo,

```
SacaCantidadMetalico(cuentaBanco, monedero, 10000)
```

reflejando el espíritu del modelo (imperativo) de programación de Pascal.
Cualquiera de las dos opciones es vÍlida, pero no deben mezclarse ambas.

- Las funciones deben nombrarse según sea el tipo de datos devuelto; de este modo, su utilización se adecuará al mismo (sustantivo o proposición).

```
inicial:= PrimeraLetra(nombre);
amortizacion:= Mensualidad(deuda, interesNominal, plazo);
REPEAT
  ...
UNTIL NOT HayFondosDisponibles(hucha, cuentaBanco)
```

3.2 Sobre el uso de constantes y variables

- Es habitual el uso erróneo de variables en un programa, por dos motivos frecuentes: uno, la presencia de variables sin asignar inicialmente, y dos, variables asignadas inicialmente sin motivo.

- El texto del programa debe evitar el uso de valores numéricos literales. La primera razón es que dificulta el mantenimiento, porque cambiar un valor literal debe hacerse minuciosamente rastreando a lo largo de todo el programa todas sus apariciones, más las de los valores que dependan de él. Otra razón es que el papel de los valores bien identificados (como `PI` por ejemplo) se interpreta mejor que sus correspondientes literales (como 0.3141592E01 por ejemplo) cuando se lee un programa.

3.3 Sobre las expresiones

- Un hábito frecuente en expresiones booleanas es compararlas con las constantes `true` y `false`:

```
IF  exprbool1 = true THEN  ...
WHILE  exprbool2 = false DO  ...
```

  Estas expresiones se debían haber escrito mejor simplemente así:

```
IF  exprbool1 THEN  ...
WHILE  NOT  exprbool2 DO  ...
```

3.4 Sobre las instrucciones estructuradas

- Nunca usar la instrucción `GOTO`.

- Al desarrollar las instrucciones de selección (condicionales y por casos), se deben contemplar todas las posibilidades de la expresión selectora. Pero esto no significa que se tengan que **incluir** siempre todas las ramas (por ejemplo, los `if`s sin `else` son lo más normal del mundo), sino que se deben haber previsto todas las situaciones posibles.

- La elección de los bucles debe responder a su cometido en el programa: número de iteraciones fijo (`FOR`), conocido *a priori*, o terminación condicionada; ninguna vuelta o más (`WHILE`) o una vuelta como mínimo (`REPEAT`).

- En los bucles `FOR`, nunca se deben modificar la variable-índice ni el límite superior.
  Tras un bucle `FOR`, el valor de la variable-índice es desconocido; por lo tanto, no se debe usar ese hipotético valor.

3.5 Sobre los subprogramas

- El uso de variables globales debe considerarse prohibido. En efecto, un subprograma que usa o modifica variables globales no está autocontenido. Por consiguiente, esta práctica ocasiona muchos errores, difíciles de localizar.
  Un remedio ideal para evitar este problema es declarar todas las variables de un programa después de los subprogramas.

- Debe evitarse el uso de subprogramas con muchos parámetros, ya que esto los hace difíciles de leer y usar. Este defecto indica usualmente un diseño pobre. Con frecuencia, este defecto se debe a usar varios parámetros que, en realidad, representan un solo dato, convenientemente agrupadas. Por ejemplo, en vez de escribir

  ```
  AvanzarFecha(numDia, diaSemana, numMes, anyo)
  ```

  podría hacerse la llamada siguiente:

  ```
  AvanzarFecha(dia)
  ```

  donde la variable `dia` es un registro que contiene todos los datos de una fecha concreta: el número de día, el día de la semana, el número del mes y el año.

- Debe evitarse el uso de subprogramas largos (más de dos páginas se considera excesivamente largo), que también son difíciles de comprender y depurar. Este defecto indica también un diseño pobre.

- Los parámetros de las funciones deberán ser siempre datos de entrada, ya sea mediante el mecanismo de paso de parámetros por valor o mediante el calificador `CONST`, que se puede aplicar a datos de gran tamaño para mejorar la eficiencia; en este último caso, estos parámetros nunca se podrán modificar en el cuerpo de la función.

- Las funciones no deben efectuar operaciones de lectura ni escritura, ya que estas operaciones producen efectos laterales sobre las variables "entrada" y "salida" de datos. En resumen, todo subprograma que lee o escribe debe ser un procedimiento.
  Por otra parte, los procedimientos que calculan nunca deben leer ni escribir, y los que escriben o leen nunca deben calcular.

- La forma en que se expresa el resultado de una función en Pascal (la asignación a la seudovariable de la función) es un error de diseño sustancial. Por consiguiente, se recomienta diseñar programas que recaigan lo mínimo posible en este diseño. Un modo usual es definir una variable `resultado`, local a la función, que almacena el resultado, y que se asigna al nombre de la función en la última línea. Otra solución es usar la variable `result`, que existe en varios dialectos de Pascal.
  El uso de una variable intermedia es el único modo de facilitar la depuración.

- En los lenguajes que permitan definir subprogramas sobrecargados (por ejemplo, freePascal permite definir operadores sobrecargados), todos los ejemplares deben responder al mismo propósito; de otro modo, se ocasionaría confusión. Por ejemplo, es posible sobrecargar el operador '+' para trabajar con números complejos o con vectores, pero sólo es adecuado hacerlo con el cometido de sumar esos tipos de datos.

3.6 Sobre los tipos de datos

- Que reflejen de forma natural las entidades del problema: enumerados mejor que codificaciones; subintervalo mejor que enteros; ...

- No usar construcciones de tipos de datos anónimos:

  ```
  VAR poligono: ARRAY[1..N] OF ARRAY[(x,y,z)] OF real;
  ```

  En su lugar, declarar las componentes e irlas identificando una a una:

  ```
  TYPE
     tEjes     = (x,y,z);
     tPunto    = ARRAY[tEjes] OF real;
     tNumPunto = 1..N;
     tPoligono = ARRAY[tNumPunto] OF tPunto;
  VAR
     poligono: tPoligono;
  ```

3.7 Sobre los comentarios

Los comentarios proporcionan la documentación del programa. Como norma general, los comentarios deben ser escuetos pero precisos, aunque pueden ser formales o informales, dependiendo de su finalidad. Explican los siguientes asuntos en relación con el programa:

- El cometido del programa, que debe reflejarse al principio del mismo.
- El papel que juegan los identificadores, cuando su identificador no baste.
- En relación con los subprogramas, debe indicarse lo siguiente:
  - Para cada parámetro, si es de entrada, de salida o ambas cosas.
  - Los requisitos de los datos (cuando no baste con su tipo), que se llaman usualmente *precondición* del subprograma.
  - Su cometido: el resultado que generan las funciones, o el trabajo que desempeñan los procedimientos. Según el caso (función o procedimiento), el cometido se expresa como el valor devuelto o la postcondición, respetivamente.

  Así pues, la especificación de un subprograma es la siguiente según el caso:

```
PROCEDURE P(...);          FUNCTION F(...): ...;
   {Pre: ...}                 {Pre: ...}
   {Post: ...}                {Dev: ...}
```

- Las instrucciones del programa y de los subprogramas se documentan del siguiente modo:
  - El cometido de un fragmento de programa se puede anticipar con un comentario:

```
BEGIN
  {Lectura de datos:}
    instrucciones para la lectura de datos
  {Cálculos:}
    instrucciones para los cálculos
  {Presentación de los resultados:}
    instrucciones de salida de resultados
END.
```

    Gries resalta el doble papel que desempeña este tipo de "instrucción comentada" (*command-comment*, en [2]): por una parte, en el refinamiento progresivo de programas, estos comentarios (seudocódigo) juegan el mismo papel que las instrucciones (código) con que se han de refinar posteriormente; por otro, cuando se lee un programa con una instrucción comentada, sólo se necesita leer el refinamiento si se desea cómo se ha desarrollado.
  - Las propiedades requeridas y garantizadas en los puntos delicados del programa. Además de las precondiciones, postcondiciones, invariantes, ya explicados, conviene a veces anotar otras propiedades con las que contamos, intercaladas entre las otras instrucciones del programa. Por ejemplo, entre varias instrucciones de una secuencia,

```
{a=A, b=B}
a:= a+b;
{a=A+B, b=B}
b:= a-b;
{a=A+B, b=A}
a:= a-b;
{a=B, b=A}
```

    ayudándonos a razonar sobre su verdadero funcionamiento; y también interesa caracterizar la situación dentro de una rama de una instrucción de selección, como en el ejemplo siguiente.
  - En particular, la documentación típica de los bucles es su invariante y su expresión cota:

```
PROCEDURE MCD({in} a, b: integer): integer;
      {Pre.: a=A, b=B, a,b>0}
      {Dev.: máximo común divisor de a y b}
   BEGIN {MCD}
```

```
   {Inv.: a,b>0, mcd(a, b)=mcd(A, B); Cota: máx(a, b)}
   WHILE a<>b DO
      IF a>b THEN
         a:= a-b
      ELSE {b>a}
         b:=b-a;
   {a=m.c.d.(a,b) = m.c.d.(A,B)}
   MCD:= a
END; {MCD}
```

3.8 Sangrado

El papel del sangrado es aclarar la estructura del texto de un modo simple y directo; esto es, qué partes están subordinadas unas a otras, dónde empiezan y terminan las distintas piezas de código (declaraciones, estructuras, módulos), etc. Como el sangrado refleja la sintaxis, las recomendaciones sobre el mismo atienden a las distintas estructuras sintácticas del lenguaje de programación.

Aunque no hay unanimidad a propósito de esto, sí un par de acuerdos de tipo general: que el cometido es reflejar la estructura, y que las reglas deben ser consistentes y aplicarse uniformemente.

Veamos algunas recomendaciones generales sobre esto en Pascal.

- En la estructura de programa, las declaraciones están subordinadas, situándose en un nivel más interno:

  ```
  Cabecera
    Declaraciones
  Cuerpo
  ```

  Véase el ejemplo global del apéndice A.

- Las instrucciones de una misma secuencia pueden ir con el mismo sangrado:

  ```
  BEGIN
    instrucción1
    instrucción2
    ...
  END
  ```

  y también pueden agruparse varias, juntas en la misma línea, cuando sean pequeños pasos de una misma acción:

  ```
  aux:= x; x:= y; y:= aux
  ```

  pero no deben colocarse en líneas distintas con distinto nivel de sangrado:

  ```
  aux:= x;
        x:= y;
     y:= aux
  ```

  Las instrucciones comentadas están al mismo nivel que las ejecutables. El refinamiento de una instrucción comentada está subordinado a ella, por lo que se sangra un nivel más:

  ```
  instrucción1
  {Intercambio de las variables a y b:}
    aux:= x;
    x:= y;
    y:= aux;
  instrucción3
  ```

- En la instrucción condicional, las dos "ramas" deben estar al mismo nivel; dos posibilidades:

```
IF condición THEN                    IF condición
  instrucción1                         THEN instrucción1
ELSE                                   ELSE instrucción2
  instrucción2
```

  Cuando las ramas sean instrucciones compuestas, los fragmentos anteriores pueden escribirse de varias formas:

```
IF condición THEN BEGIN      IF condición            IF condición THEN BEGIN
  instrucciones                THEN  BEGIN             instrucciones
END {IF}                         instrucciones       END ELSE  BEGIN
ELSE  BEGIN                    END {IF}                  instrucciones
  instrucciones                ELSE  BEGIN           END {ELSE}
END {ELSE}                       instrucciones
                               END {ELSE}
```

- En los bucles incondicionales, el cuerpo es la instrucción subordinada:

```
FOR indice:= valor inicial TO valor final DO
  instrucción
```

  aunque, cuando se trata de una secuencia, no hace falta sangrar dos veces:

```
FOR indice:= valor inicial TO valor final DO BEGIN
  instrucción1;
  instrucción2;
  ...
END; {FOR}
```

- Los bucles condicionales siguen el mismo criterio:

```
WHILE condición DO    WHILE condición DO BEGIN    REPEAT
  instrucción;          instrucción1;               instrucción1;
                        instrucción2;               instrucción2;
                        ...                         ...
                      END; {WHILE}                UNTIL condición
```

- Subprogramas: como los programas, las declaraciones están subordinadas, situándose en un nivel más interno:

```
Cabecera
  Declaraciones
Cuerpo
```

- Finalmente, no es conveniente un sangrado excesivo, para no rebasar el ancho usual (80 columnas) del monitor o del papel, trabajando con texto estándar.

3.9 Convenciones sobre el uso de mayúsculas y minúsculas

El uso de mayúsculas y minúsculas también ayuda a leer y comprender mejor los programas, ya que permite distinguir mejor diferentes elementos léxicos, como las palabras reservadas o los identificadores de distinto tipo. En principio, no hay razón alguna para preferir uno u otro convenio, pero sí interesa escoger uno por uniformidad y coherencia.

Un posible conjunto de recomendaciones es el siguiente, que es el que se ha seguido en los ejemplos de este folleto:

- Usar mayúsculas para el comienzo de las constantes y subprogramas.
- Usar minúsculas para el comienzo de las variables.
- Marcar los identificadores de tipos inicialmente con una '`t`'.
- Cuando un identificador conste de varias palabras, se recomienda poner con mayúscula el comienzo de la segunda, tercera, etc., para facilitar su lectura:

  ```
  LeerDatos...
  ConvertirGradosEnRadianes...
  ```

  Durante largo tiempo, se ha venido usando también el carácter "`_`" (guión bajo) para separar distintas palabras en un mismo identificador,

  ```
  Leer_datos...
  Convertir_grados_en_radianes...
  ```

  aunque este carácter no es válido en Pascal estándar.

En cuanto a las palabras reservadas, es costumbre en Pascal distinguirlas, bien con negrita (en los libros) o subrayándolas (cuando se escribe a mano), para resaltar su papel sintáctico, que facilita la lectura. El inconveniente de estas alternativas es que ninguna de ellas usa texto estándar, de forma que al escribir los programas en un editor tenemos que pensar en otra posibilidad. Por eso, las palabras reservadas también se escriben completamente en mayúsculas, como estamos haciendo en este manualillo.

3.10 Convenciones a propósito de los tipos abstractos de datos (TADs)

- Todo uso de un TAD debe efectuarse siempre mediante sus operaciones propias.
  Posiblemente, el error más frecuente del usuario de TADs es conocer su implementación y hacer uso de ella. Este defecto suprime la abstracción, lo que contradice evidentemente el objetivo de los TADs.
- La documentación necesaria para los módulos de TADs, consiste en lo siguiente:
  - Los requisitos de los parámetros del TAD.
  - La especificación completa de su comportamiento.
  - El coste de cada operación.
  - La implementación elegida: estructuras de datos e invariante de representación.

Referencias

[1] R. A. di Falco, *xBase Identifier Naming Conventions -or- HUNG Hungarian types without the arian*, manuscrito, revision 1.2, 1994.

[2] D. Gries, *The Science of Programming*, Springer-Verlag, 1981, capítulo 22.

[3] D. E. Knuth, Literate Programming, Center for the Study of Language and Information, Lecture Notes Number 27, 1992.

[4] B. J. Poole y T. S. Meyer, Implementing a Set of Guidelines for CS Majors in the Production of Program Code, *SIGCSE Bulletin*, vol. 28(3), jun. 1996, 43–48.

[5] L. Rising, Teaching Documentation and Style in Pascal, *SIGCSE Bulletin*, vol. 19(3), sep. 1987, 8–14.

A Apéndice: Ejemplos completos

A.1 Fecha correcta

```
PROGRAM ComprobacionFechas (input, output);
   {Pide repetidamente fechas hasta que se dé una correcta por el teclado}
   TYPE
      tFecha = RECORD
         dia, mes, anyo: integer
      END; {tFecha}

   FUNCTION EsCorrecta ({in} fecha: tFecha): boolean;
      {Pre: No hay requisitos
       Dev: True si (dia,mes,anyo) de la fecha dada es una fecha correcta;
            False en caso contrario}

      FUNCTION NumDias({in} mes, anyo: integer): integer;
         {Pre: 1<=mes<=12 anyo>=1800
          Dev: el número de días que tiene el mes en el anyo indicado}

         FUNCTION EsBisiesto({in} anyo: Integer): Boolean;
            {Pre: anyo>=1800
             Dev: True si el año es bisiesto;
                  Falso en caso contrario}
         BEGIN {EsBisiesto}
            IF anyo MOD 400 = 0 THEN
               EsBisiesto:= True
            ELSE IF anyo MOD 100 = 0 THEN {anyo MOD 100=0 y anyo MOD 400<>0}
               EsBisiesto:= False
            ELSE {anyo MOD 100<>0 y anyo MOD 400<>0}
               EsBisiesto:= anyo MOD 4 = 0
         END; {EsBisiesto}

      BEGIN {NumDias}
         CASE mes OF
           1,3,5,7,8,10,12: NumDias:= 31;
           4,6,9,11       : NumDias:= 30;
           2              : IF EsBisiesto(anyo) THEN
                               NumDias:= 29
                            ELSE
                               NumDias:= 28
         END {CASE}
      END; {NumDias}

   BEGIN  {EsCorrecta}
      WITH fecha DO
         esCorrecta:= (anyo>=1800) AND (mes>=1) AND (mes<=12) AND
                      (dia>=1) AND (dia<=NumDias(mes,anyo))
   END; {EsCorrecta}

   VAR
      fecha: tFecha;

BEGIN {ComprobacionFechas}
   REPEAT
      WriteLn('Dame la fecha: ');
      WITH fecha DO
         ReadLn(dia,mes,anyo)
   UNTIL EsCorrecta(fecha)
END. {ComprobacionFechas}
```

A.2 Fecha correcta

El siguiente programa es correcto desde el punto de vista del funcionamiento, e incluso está bien documentado. Sin embargo, se presenta como programa inaceptable por razones obvias:

```
PROGRAM ComprobacionFechas (input, output); {Pide repetidamente fechas
hasta que se dé una correcta por el teclado} TYPE tFecha = RECORD
dia, mes, anyo: integer END; {tFecha} FUNCTION EsCorrecta ({in}
fecha: tFecha): boolean; {Pre: No hay requisitos Dev True si (dia,
mes,anyo) de la fecha dada es una fecha correcta; False en caso
contrario} FUNCTION NumDias({in} mes,anyo: integer):integer; {Pre:
1<=mes<=12 anyo>=1800 Dev: el numero de dias que tiene el mes en el
anyo indicado} FUNCTION EsBisiesto({in} anyo: Integer): Boolean;
{Pre: anyo>=1800 Post: La función devuelve True si el año es bisiesto;
Falso en caso contrario} BEGIN {EsBisiesto} IF anyo MOD 400 = 0 THEN
EsBisiesto := True ELSE IF anyo MOD 100 = 0 THEN {anyo MOD 100=0 y anyo
MOD 400<>0} EsBisiesto := False ELSE {anyo MOD 100<>0 y anyo MOD 400<>0}
EsBisiesto := anyo MOD 4 = 0 END; {EsBisiesto} BEGIN {NumDias} CASE
mes OF 1,3,5,7,8,10,12 : NumDias := 31; 4,6,9,11 : NumDias := 30; 2 :
IF EsBisiesto(anyo) THEN NumDias := 29 ELSE NumDias := 28 END {CASE}
END; {NumDias} BEGIN {EsCorrecta} WITH fecha DO esCorrecta :=
(anyo>=1800) AND (mes>=1) AND (mes<=12) AND (dia>=1) AND
(dia<=NumDias(mes,anyo)) END; {EsCorrecta} VAR fecha: tFecha; BEGIN
{ComprobacionFechas} REPEAT WriteLn('Dame la fecha: '); WITH fecha DO
ReadLn(dia,mes,anyo) UNTIL EsCorrecta(fecha) END. {ComprobacionFechas}
```

A.3 Mínimo elemento de un conjunto

El siguiente fragmento de programa ejemplifica el uso de propiedades, formalmente, para especificar un subprograma y un bucle.

```
CONST
  Min= 1; Max= 50;

TYPE
  tNaturales= Min..Max;
  tConjuntoNaturales= SET  OF  tNaturales;

FUNCTION  MinimoElemento({in} conjunto:  tConjuntoNaturales):  tNaturales;
  {Pre:  conjunto ≠ ∅
   Dev:  n ∈ conjunto.(∀k < n.k ∉ conjunto)}
  VAR
    n:  tNaturales;
BEGIN  {MinimoElemento}
  n:= Min;
  {Inv:  ∀k < n.k ∉ conjunto ∧ ∃k ≥ n.k ∈ conjunto.  Cota:  Max-n}
  WHILE  NOT  (n IN  conjunto) DO
    n:= n+1;
  {n ∈ conjunto ∧ (∀k < n.k ∉ conjunto)}
  MinimoElemento:= n
END; {MinimoElemento}
```

Las propiedades anteriores pueden expresarse también informalmente, aunque lo importante es que se haga con la misma precisión y claridad:

```
CONST
   Min= 1; Max= 50;

TYPE
   tNaturales = Min..Max;
   tConjuntoNaturales = SET OF tNaturales;

FUNCTION MinimoElemento({in} conjunto: tConjuntoNaturales): tNaturales;
     {Pre: conjunto no es vacío
      Dev: el mínimo elemento de conjunto}
   VAR
      n: tNaturales;
BEGIN {MinimoElemento}
   n:= Min;
   {Inv: ningún elemento menor que n está en el conjunto,
    y a partir de n sí hay algún elemento en conjunto}
   WHILE NOT (n IN conjunto) DO
      n:= n+1;
   {n es el mínimo elemento del conjunto}
   MinimoElemento:= n
End; {MinimoElemento}
```