MANUAL DEL USUARIO DEL LENGUAJE UBL

Documento Numero UBL-MU-001-0

3 de Julio de 1985

Jose Maria Blasco Comellas

Centro de Informatica
Universidad de Barcelona

Manual del Usuario del Lenguaje UBL

Indice

Lista de Figuras

Lista de Algoritmos

Introduccion

La multiplicacion de Gitanos, version 1

Cuentan que algunos vendedores ambulantes, perezosos para aprender a multiplicar, utilizan un metodo simplificado, que solo requiere saber sumar, doblar y tomar mitades.

Ilustraremos el metodo con un ejemplo.

(1)	*(2)*	*(3)*	*(4)*	*(5)*
23 45	*23 45*	*23 45*	*23 45*	*23 45*
	46 22	*46 22*	~~*46 22*~~	~~*46 22*~~
		92 11	*92 11*	*92 11*
		184 5	*184 5*	*184 5*
		368 2	~~*368 2*~~	~~*368 2*~~
		736 1	*736 1*	*736 1*
				1035

Figura 1. Metodo de la multiplicacion de Gitanos

Supongamos que queremos multiplicar 23 y 45. Primero los ponemos uno al lado del otro (1). En la linea siguiente escribimos: debajo del primero, el doble, y debajo del segundo, la mitad (ignorando decimales, si los hay [2]), y repetimos el proceso hasta que obtengamos un 1 en la columna de la derecha (3). Despues tachamos las filas que tengan un numero par en la segunda columna (4), y sumamos los numeros que quedan en la primera columna (5). El resultado de la suma es el producto deseado (el metodo puede aplicarse a cualquier otro par de numeros naturales).

Esta forma de multiplicar, que tecnicamente se conoce como *multiplicacion binaria*, es atribuido por algunos a los mercaderes gitanos, y por otros, a los rusos.

La multiplicacion de Gitanos, version 2

Otra version del mismo metodo consiste en bajar tres columnas en vez de dos, acumulando en la tercera la suma de la primera cuando sea necesario (esto es, cuando la segunda sea impar):

```
23 45   23
46 22   23
92 11  115
184 5  299
368 2  299
736 1 1035
```

Aunque hay que escribir mas, de este modo no es necesario volver sobre los resultados anteriores; de hecho, una vez calculada una fila, puede prescindirse de la anterior. El resultado se encuentra al final de la tercera fila.

Lenguajes Naturales y Artificiales. Lenguajes de Programacion

Estas explicaciones puede ser suficientes para que una persona comprenda el metodo y pueda aplicarlo (siempre que entienda los conceptos que se dan por supuestos, como la suma o tomar mitades), pero, en el estado actual de la tecnologia, no son comprensibles directamente por un

ordenador. Es sabido que el lenguaje escrito (y tambien el hablado) suele ser fuente de ambiguedades (la frase "Adios" puede ser una despedida o una invitacion a marcharse, segun el tono y la circunstancia): los ordenadores no pueden decidir de que se esta hablando como lo hace una persona.

Asi, suelen utilizarse en la mayoria de los ordenadores lenguajes artificiales o *de Programacion* (que llamaremos aqui simplemente *lenguajes*), mas sencillos y menos ambiguos en su estructura que el lenguaje hablado. En la mayoria de los casos, suelen ser lenguajes mixtos entre la notacion matematica y el lenguaje habitual. Las hipotesis implicitas al utilizar estos lenguajes varian con cada uno de ellos, asi como el modelo conceptual sobre el que se construyen las instrucciones al ordenador. Aqui utilizaremos el Lenguaje de la Universidad de Barcelona (UBL), que permite escribir programas en una notacion mas proxima al lenguaje hablado que otros lenguajes.

Metodos o algoritmos; Programas

La idea principal es que proporcionamos al ordenador "metodos para hacer cosas"; a estos metodos los llamamos *algoritmos*, y a sus descripciones utilizando determinada notacion las llamamos *programas*.

En el ejemplo mostrado, deseamos "enseñarle" al ordenador a realizar la multiplicacion de Gitanos. Queremos que lo aprenda *en general*, esto es, que sea capaz de aplicarlo para multiplicar cualquier par de numeros, y no solo 23 y 45. Para ello, deberemos construir un programa (una descripcion del metodo), de acuerdo a las convenciones del lenguaje utilizado.

Desarrollamos a continuacion un programa en el lenguaje UBL para describir la version 2 de la multiplicacion de Gitanos; el proposito de esta exposicion es proporcionar una impresion general de como se trabaja con el lenguaje: muchos conceptos se entenderan solo aproximadamente, y algunos detalles sintacticos (como la presencia de puntos y coma o la palabra **fin**) podran parecer arbitrarios. Todo ello se explicara detalladamente en capitulos posteriores.

Construccion de un programa para la multiplicacion de Gitanos

Identifiquemos cada columna de la tabla con una letra:

m	*n*	*p*
23	45	23
46	22	23
92	11	115
184	5	299
368	2	299
736	1	1035

Las transformaciones necesarias para obtener una linea a partir de la anterior pueden expresarse precisamente mediante

dobla el valor de la columna m
toma la mitad en la columna n
si n *es impar, suma* m *a* p

En el lenguaje UBL, se escribira

```
m ← 2 * m;
n ← n div 2;
si impar(n) entonces p ← p + m; fin;
```

La notacion utilizada requiere cierta explicacion: *2 * m* significa "el producto de 2 por *m*" (el simbolo "*" substituye a las aspas de multiplicar en la mayoria de ordenadores). "**div**" significa "dividido por"; en cuanto al simbolo "←", que se lee "toma por valor", es el modo de dar valor a un identificador. A la operacion de dar valor la llamaremos *asignacion*.

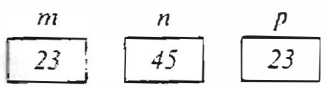
Una vez expresado el paso de una fila a otra, el proceso entero puede resumirse como

pasa de una fila a otra hasta que n *sea igual a 1*

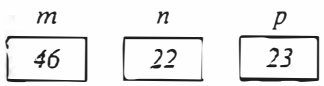
que se expresara en UBL del siguiente modo

```
repite
  m ← 2 * m;
  n ← n div 2;
  si impar(n) entonces p ← p + m; fin;
hastaque n = 1;
```

M, *n* y *p*, entidades a las que llamaremos *variables*, pueden verse como pizarras en las que podemos escribir numeros enteros. A partir de una fila



pasamos a la siguiente modificando el valor o contenido de estas variables



y asi sucesivamente. Es necesario informar al ordenador de cuantas variables necesitamos, que nombres tienen y que tipo de informacion van a contener: esto se hace mediante la *declaracion*

```
var m,n,p: entero;
```

Inicialmente (al empezar el proceso), las variables deberan ser "llenadas" con los valores apropiados (esto es, *m* y *n*, con los valores que se desee multiplicar, y *p* con *0* o el valor de *m* segun *n* sea impar). Hemos dicho que explicamos a la maquina el metodo general; cada utilizacion (o *ejecucion*) del metodo debera hacerse con un par concreto de valores iniciales: el ordenador pedira estos valores (por la pantalla u otro medio) mediante una operacion que llamaremos *lectura*:

```
lee m,n;
si impar(n) entonces p ← m; sino p ← 0; fin;
```

Finalmente, sera necesario que el ordenador nos informe del resultado de la multiplicacion: paralelamente a la operacion de lectura, que sirve para dar valor a variables a partir de datos externos al programa, utilizaremos la operacion de *escritura*

```
escribe p;
```

Asi, el programa completo se escribira:

```
programa Multiplicacion_de_Gitanos es
  var m,n,p: entero;
haz
  lee m,n;
  si impar(n) entonces p ← m; sino p ← 0; fin;
  repite
    m ← 2 * m;
    n ← n div 2;
    si impar(n) entonces p ← p + m; fin;
  hastaque n = 1;
  escribe p;
fin programa;
```

Algoritmo 1. Multiplicacion de Gitanos.

La diferente posicion (o *indentacion*) de las lineas con respecto al margen izquierdo se utiliza para señalar la subordinacion logica de los instrucciones empleadas: "*m* ← *2* * *m*" es una de las instrucciones que deben repetirse, y por ello se encuentra mas a la derecha que la palabra **repite**.

Discusion

Se han presentado los conceptos principales utilizados en la mayoria de lenguajes de programacion: las *variables* como pizarras o cajas que pueden contener informacion de determinado tipo, y que deben definirse o *declararse*; la *asignacion*, que permite alterar el valor de estas variables; la *seleccion* o *toma de decision* (instruccion **si**), que permite que el ordenador obedezca una instruccion u otra segun se cumpla o no determinada condicion; y la *iteracion* (instruccion **repite**), que permite realizar repetidamente un grupo de instrucciones.

Hemos elegido la version 2 del metodo para escribir el programa, ya que la version 1, al requerir volver sobre los resultados anteriores (p. ej., tachando columnas) para obtener el resultado, es dificil de programar directamente sin recurrir a un conjunto potencialmente infinito de variables (ya que no es previsible, sin conocer los valores de m y n, el numero de lineas que se produciran, y no hay manera, sin tener una variable para cada numero, de "recordar" los valores de estas variables). Esto es frecuente en programacion: un metodo puede ser adecuado para realizar a mano, pero una version aparentemente mas complicada puede ser mas util o simple en un ordenador.

Ejercicios

1. Puede ser interesante que el programa proporcione alguna informacion sobre lo que se desea, y tambien que, ademas del resultado final, escriba todos los pasos del calculo. Sabiendo que es posible utilizar la instruccion *escribe_linea*, que escribe (posiblemente, presenta por pantalla) datos, en la forma

   escribe_linea "Cualquier texto entre comillas";

   o bien

   escribe_linea m,n,p;

   modificar el Algoritmo 1 para que realice estas operaciones.

2. Mostrar que, esencia, el metodo se reduce a una multiplicacion simple en base 2.

Conceptos Basicos

Algoritmos y programas.

Un *algoritmo* es un metodo para realizar algun tipo de proceso (de un modo mas tecnico, es una clase de trasformaciones de datos); un *programa* es la especificacion de un algoritmo en determinado lenguaje. Los programas suelen escribirse con el proposito de que sean *interpretados* o *ejecutados* por un ordenador: esta interpretacion consiste en la ejecucion de las *instrucciones* que el programa especifica.

Pocos programas se escriben para realizar un proceso concreto; la mayoria de ellos aceptan *datos*, que actuan como parametros del proceso. Podemos verificar que un programa "funciona", esto es, que sus especificaciones responden a nuestro proposito al escribirlo, para unos datos concretos; verificarlo para todos los datos posibles es, en general, una tarea inviable, de modo que la *correccion* de un programa debera establecerse a partir de un diseño y analisis cuidadosos del mismo.

Compilacion

Un programa se escribe, y se suministra al ordenador, en forma de texto. El ordenador verifica que el texto este escrito conforme a las reglas que prescribe el lenguaje, detectando construcciones erroneas. Al mismo tiempo, "comprende" el programa, y posiblemente lo traduzca a un formato interno mas sencillo para su posterior interpretacion. A este doble proceso lo llamamos *compilacion*, y *compilador* al agente que lo realiza. La comprension del texto (por el ordenador) solo es posible si este no contiene *errores de compilacion* (en cuanto a las reglas de escritura del lenguaje); en caso de que los haya, el compilador informa de ellos mediante algun tipo de mensaje: la mayoria de compiladores producen un *listado*, que suele contener, ademas del programa (posiblemente "embellecido" con algun mecanismo, o incluso reescrito para aumentar su legibilidad), informacion adicional (numeracion de las instrucciones, estadisticas de compilacion, errores detectados, etc).

Errores logicos

Hay que notar que una compilacion correcta (esto es, en la que no se han detectado errores) no garantiza que el programa funcione como deseamos: es posible (tanto en lenguajes naturales como artificiales) escribir un texto sintacticamente correcto que no signifique nada, o haber expresado mal lo que queremos decir. Si un programa bien compilado funciona mal, diremos que contiene un *error logico*: en este caso, no hay ningun mecanismo que nos informe de las causas del error (ya que no hay nadie, aparte de nosotros mismos, que pueda saber que intentabamos decir al escribir un programa). La correccion de los errores logicos es generalmente mucho mas complicada que la de los de compilacion, y a veces requiere de mecanismos auxiliares. Al conjunto de acciones que la constituyen se le suele llamar *depuracion* de un programa.

Fases del diseño y elaboracion de un programa

Para programas sencillos, el proceso (ideal) necesario para la obtencion de un programa efectivo puede resumirse mediante los siguientes pasos:

- *Analisis y delimitacion teoricos* del problema, que debera realizarse sobre bases lo mas abstractas posible, sin referencia concreta al ordenador.
- *Escritura de un programa* de acuerdo con las especificaciones del problema definidas en la fase anterior. Hay varias aproximaciones a este proceso, que se comentaran en capitulos posteriores.

- *Sometimiento del programa escrito al compilador.* Si se detectan errores en esta fase, sera necesario corregirlos y repetir la compilacion.
- *Prueba* del programa con una muestra de datos razonablemente amplia. Si se producen fallos, sera necesario revisar el programa y volver a compilar, o incluso revisar el modelo teorico. *Depuracion* en este caso para detectar con exactitud las partes incorrectas del programa.

que tambien pueden presentarse en forma de algoritmo en UBL:

```
diseño_teorico;
escritura_del_programa;
compilacion;
si hay_errores entonces
  repite
    correccion_de_los_errores;
    compilacion;
  hastaque no hay_errores;
fin si;
prueba;
si no funciona entonces
  repite
    depuracion_e_identificacion_de_los_errores;
    correccion_de_los_errores;
    compilacion;
    prueba;
  hastaque funciona;
fin si;
```

Algoritmo 2. Fases del diseño y elaboracion de un programa

El modelo de ejecucion secuencial

Supondremos que cada programa define un *orden* estricto en la ejecucion de las instrucciones que componen el programa; una instruccion se obedece, efectua o *ejecuta* realizando la *accion* que designa; tal ejecucion tiene un *efecto* sobre el *estado* del programa, efecto que podemos tomar en consideracion al ejecutar la siguiente instruccion. Dicho mas sencillo, cada instruccion se ejecuta completamente antes de ejecutar la siguiente. Notese que el orden de ejecucion de las instrucciones no tiene porque corresponder con el orden textual de estas: por ejemplo, una instruccion repetitiva determina la ejecucion continuada del mismo grupo de instrucciones.

Objetos

Las instrucciones que contiene un programa pueden afectar a entidades que llamaremos *objetos*, y que suelen representar abstracciones, tomadas del "mundo real" o de determinado modelo conceptual: en el Algoritmo 1 en la pagina 3, *m*,*n* y *p* son objetos. Puede pensarse en un objeto como en una pizarra: cada pizarra tiene su *nombre* (que llamaremos a veces *identificador*), esta limitada a contener informacion de determinado *tipo*, y posiblemente contiene un *valor*:

puede ser un objeto de nombre *n*, tipo *entero* y valor *23*.

Hay dos operaciones basicas aplicables a los objetos, que corresponden a las ideas intuitivas de consultar y alterar el valor escrito en una pizarra: las llamaremos *inspeccion* y *modificacion*. Convendremos en que [el valor de] cualquier objeto podra ser inspeccionado, pero no todos los objetos podran ser modificados: llamaremos *variables* a los objetos modificables, y *constantes* a los que no lo son.

En el caso de construcciones como *23* o *-12* convendremos en que son objetos constantes, cuyo nombre identifica su valor, y los llamaremos *literales*.

Elementos del lenguaje UBL

Instruccion de asignacion. Expresiones y evaluacion

La instruccion mas sencilla es la de *asignacion*:

---

variable ← *expresion;*

Figura 2. Sintaxis de la instruccion de asignacion

El simbolo '←' se lee "toma por valor", y se llama *operador de asignacion.* El efecto de una asignacion consiste en la evaluacion de la expresion (o *parte derecha* de la asignacion) y la modificacion del valor de la variable (o *parte izquierda*) para ser el resultado de la expresion.

Suponiendo la existencia de una variable *v* de tipo *entero* y valor arbitrario, la ejecucion de

v ← *3;*

tiene como efecto que el valor de *v* pase a ser 3.

Entendemos por *expresion* cualquier formula, escrita siguiendo las convenciones del lenguaje, que seran descritas mas adelante. Por el momento, bastara saber que una expresion puede contener tanto literales como otros objetos, tomandose estos ultimos por una especificacion de su valor; podran realizarse las operaciones aritmeticas usuales, utilizando parentesis cuando sea necesario. Al manejar numeros enteros, se dispondra de los operadores de adicion (+), substraccion (-), multiplicacion (*), division o cociente (**div**) y modulo (**mod**). Nos referiremos al calculo del valor de una expresion hablando de su *evaluacion.*

Iteracion. Instruccion repite

En la mayoria de los lenguajes *imperativos* (que utilizan instrucciones) existen instrucciones *iterativas* o repetitivas, que se utilizan para efectuar la ejecucion repetida de otra instruccion. La idea es que un proceso puede hacerse por partes, realizando en cada repeticion una parte del proceso total. Asi pues, es necesario que cada ejecucion de la instruccion repetida nos "acerque" en cierto sentido a la consecucion del objetivo propuesto.

Se ha visto un ejemplo de instruccion iterativa (**repite**) en el Algoritmo 1 en la pagina 3. La forma general de la instruccion es

---

repite
  instruccion(es)
hastaque *condicion;*

Figura 3. Sintaxis de la instruccion **repite**

---

donde *condicion* significa cualquier expresion que [tenga un valor que] sea *cierto* o *falso*, p. ej., una comparacion del estilo de $n = 1$.

El efecto de la ejecucion de esta instruccion, de acuerdo con su significado intuitivo, consiste en ejecutar la(s) instruccion(es) subordinadas, evaluar la condicion, y, si su resultado es *falso*, repetir el proceso hasta que sea *cierto.*

Es importante darse cuenta de que esto no significa que si la condicion se cumple a mitad de la ejecucion de la instruccion subordinada su ejecucion se detenga: en

```
repite
  a ← 2;
  b ← b - 1;
hastaque a = 2;
```

aunque la ejecucion de $a \leftarrow 2$ hace que la condicion $a = 2$ se cumpla, esta no se evalua hasta que finaliza la ejecucion de $b \leftarrow b-1$.

Seleccion. Instruccion si

Otro tipo de instruccion universalmente utilizado es la *seleccion* o *toma de decision*, que supone en el interprete del lenguaje la facultad de ejecutar una instruccion u otra segun se verifique determinada condicion. El ejemplo mas sencillo de instruccion de seleccion es la instruccion **si**, de la que ya se han visto varios ejemplos.

---

si *condicion* **entonces**
  instruccion(es)$_1$
sino
  instruccion(es)$_2$
fin si;

Figura 4. Sintaxis de la instruccion **si**

---

cuyo efecto consiste en evaluar la condicion, y ejecutar la(s) instruccion(es)$_1$ o la(s) instruccion(es)$_2$ segun sea *cierto* o *falso*, respectivamente, el resultado.

"**Sino** instruccion(es)$_2$" puede omitirse si no se desea efectuar nada en el caso de que sea falsa la condicion; y el **si** de "**fin si**;" puede ser omitido (aunque se aconseja que solo se haga en instrucciones textualmente muy cortas).

Obtencion y presentacion de datos: entrada y salida

Los datos que un programa necesita se suministran al ordenador a partir de algun medio externo al universo en el que el programa se ejecuta (p. ej., la terminal); igualmente, es necesario presentar los resultados de un proceso. El lenguaje UBL define instrucciones (llamadas por tradicion *instrucciones de entrada y salida*) para realizar estas operaciones:

lee variables;

obtiene a partir de algun medio externo (posiblemente la terminal) los valores de las variables, que deben separarse con comas. La lectura de variables de tipo entero requiere tan solo la aparicion de un numero entero en el medio externo, posiblemente precedido de cualquier numero de espacios en blanco; la lectura de caracteres se realiza caracter por caracter, contando el espacio en blanco como un caracter como los demas.

escribe expresiones;

evalua las expresiones, que deben separarse con comas, y las presenta en algun medio externo (posiblemente la terminal). Las expresiones pueden reducirse a literales o variables, y tambien pueden incluirse textos encerrados entre comillas (que son literales del tipo *tira*, que se estudiara mas adelante).

Tipos: *caracter* y *entero*

El lenguaje define varios *tipos* de datos, ademas de la posibilidad de crear nuevos tipos (como se vera en capitulos posteriores). Se han tratado ya objetos de tipo *entero*; definiremos ahora un nuevo tipo, que llamaremos *caracter*: un objeto de tipo *caracter* podra tener como valor cualquier letra (mayuscula o minuscula), numero o signo de puntuacion (incluyendo el espacio en blanco, parentesis, llaves, corchetes y posiblemente otros, dependiendo del ordenador). Los literales de tipo *caracter* se escriben entre apostrofes: *'A'*, *'a'*, *'2'*, *' '* o *'%'* son caracteres. Se denota por *''''* el caracter cuyo valor es el apostrofe.

Declaraciones. Declaraciones de objetos

Las entidades que se manipulan en un programa (objetos u otros tipos de entidad que se estudiaran) necesitan ser definidas o *declaradas* antes de utilizarse. En tales definiciones se especifica la naturaleza y propiedades de la entidad, de modo que cualquier uso incoherente con su declaracion pueda ser detectado por el compilador.

Las variables se declaran especificando su tipo:

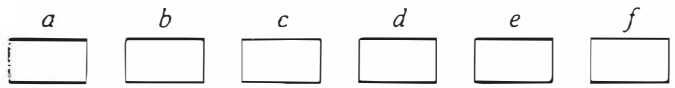
```
var identificador(es) : tipo.
```

Figura 5. Declaracion de **variables**

define variables asociadas a los identificadores (se separaran por comas si hay varios). La palabra **var** es opcional, aunque recomendamos suprimirla solo en declaraciones sucesivas que quepan en una sola linea:

```
var a,b,c: entero; d,e,f: caracter;
```

define seis objetos de nombres *a*, *b*, *c*, *d*, *e* y *f*, con tipos *entero* (los tres primeros) y *caracter*. Esta declaracion es necesaria para que el interprete "cree" las variables y podamos trabajar con ellas:



Una declaracion de variable no da valor a la variable; diremos que su valor queda *indefinido* hasta que se le asigne alguno directamente (por ejemplo, mediante una *inicializacion* [ver mas adelante]).

Una constante se declara especificando su valor (que sera suficiente para conocer su tipo):

```
const identificador = valor.
```

Figura 6. Declaracion de **constantes**

Valor puede ser un literal, otra constante o una expresion que involucre solo constantes:

```
const max = 100;
const max2 = max * max;
```

define dos objetos constantes de nombres *max* y *max2*, tipo *entero* (implicito en el valor de *max*) y valores *100* y *100000* respectivamente.

Forma general de un programa en UBL

```
programa nombre_del_programa es
  declaraciones
haz
  instrucciones
fin programa;
```

Figura 7. Sintaxis de un **programa**

Se notara que se separan (conceptual y textualmente) las declaraciones de las instrucciones mediante la palabra **haz**.

Un ejemplo: programa para contar las letras A

Definicion del problema: Nos proponemos escribir un programa que tome como dato una frase terminada por un punto y cuente el numero de letras A que contiene. Para simplificar, contaremos solo las As mayusculas, y supondremos que la frase contiene al menos una letra.

Analisis de una posible solucion: Analizaremos los caracteres (letras) de la frase uno por uno, en el orden en el que se presentan, hasta llegar al punto que la termina; y anotaremos cada ocurrencia de la letra A mayuscula, hasta obtener el numero de sus apariciones.

Diseño de un algoritmo: Definiremos dos variables, que contendran el caracter que estamos examinando y el numero de letras A encontradas:

```
var c: caracter; n: entero;
```

Obtendremos los caracteres de la frase mediante la ejecucion repetida de

```
lee c;
```

Cuando encontremos una letra A, incrementaremos el valor de *n*, para dejar constancia de su aparicion:

```
n ← n + 1;
```

El proceso debera repetirse hasta encontrar el punto que termina la frase:

```
repite
  lee c;
  si c = 'A' entonces n ← n + 1; fin;
hastaque c = '.';
```

Obviamente, sera necesario indicar que, antes de examinar cualquier caracter, el numero de apariciones de la letra A es cero

```
n ← 0;
```

Si añadimos instrucciones para informar de que proceso realiza el programa y de cual es el resultado

```
escribe_linea "Escribe una frase terminada por un punto:";
```

y

```
escribe_linea "El numero de letras A que hay en esta frase es ",n;
```

obtenemos el programa completo:

```
programa Cuenta_las_As es
  var c: caracter; n: entero;
haz
  escribe_linea "Escribe una frase terminada por un punto:";
  n ← 0;
  repite
    lee c;
    si c = 'A' entonces n ← n + 1; fin;
  hastaque c = '.';
  escribe_linea "El numero de letras A que hay en esta frase es ",n;
fin programa;
```

Algoritmo 3. Contar las As de una frase terminada por un punto

Discusion:: la iteracion es correcta: en primer lugar, nos aproximamos al fin del proceso (ya que la instruccion *lee c* avanza conceptualmente hacia el fin de la frase); en segundo lugar, el proceso termina, ya que hemos supuesto que la frase termina con un punto, lo cual queda reflejado en la condicion de terminacion $c='.'$.

La asignacion $n \leftarrow 0$ es necesaria para que se verifique la asercion inicial de que, antes de examinar ningun caracter, no hemos hallado ninguna letra A. Este tipo de asignaciones preliminares suelen llamarse *inicializaciones*.

Ejercicios

1. Algunos lenguajes definen una instruccion de *asignacion multiple* que tiene como efecto, en este orden, la evaluacion de las expresiones de la parte derecha de la asignacion y la asignacion de los valores resultantes a las variables de la parte izquierda:

   ```
   x, y ← 2, 3;
   ```

   tiene el mismo efecto que

$x \leftarrow y$; $y \leftarrow x$

Para intercambiar el valor de dos variables x e y podra hacerse

$x, y \leftarrow y, x$;

Escribir instrucciones en UBL (que no proporciona esta posibilidad) que realicen este intercambio. [**Indicacion**: disponer, si se considera necesario, de variables adicionales o auxiliares].

2. Escribir un programa que, a partir de una serie de numeros no vacia terminada con un cero, cuente la cantidad de numeros pares que contiene.

3. Escribir un programa que, a partir de una serie de numeros terminada por un cero, proporcione la media aritmetica de estos numeros.

Operadores logicos

Condiciones complejas y operadores logicos

Recordaremos que hemos llamado *condiciones* a las expresiones que pueden ser ciertas o falsas (Como se vera, estas expresiones tienen su tipo: el tipo *logico*, que sera estudiado mas adelante). En muchas ocasiones es necesario utilizar varias condiciones a la vez, ya sea en forma de conjuncion, disyuncion o negacion. Definiremos tres operadores **o**, **y** y **no**, aplicables entre o sobre expresiones que puedan valer *cierto* o *falso* (esto es, que sean de tipo *logico*), del mismo modo que en la logica tradicional:

A **o** *B*	es *cierto*	si *A* o *B* son *cierto*
A **y** *B*	es *cierto*	solo si *A* y *B* son *cierto*
no *A*	es *cierto*	si *A* es *falso*

lo que tambien puede expresarse mediante las siguientes tablas, llamadas "tablas de verdad":

A	*B*	*A* **y** *B*	*A* **o** *B*	**no** *A*
Cierto	*Cierto*	*Cierto*	*Cierto*	*Falso*
Cierto	*Falso*	*Falso*	*Cierto*	*Falso*
Falso	*Cierto*	*Falso*	*Cierto*	*Cierto*
Falso	*Falso*	*Falso*	*Falso*	*Cierto*

Figura 8. Tablas de verdad para los operadores logicos

Estos operadores pueden aplicarse para verificar el cumplimiento de condiciones complejas, que involucren inspecciones de varias variables al mismo tiempo

```
a = 3 y b = 5
c = 'a' o c = 'A'
no (n = 3 y m = -3)
```

y "funcionan" conforme a su significado intuitivo.

Los operadores **y** y **o** son asociativos (cada uno de ellos: puede escribirse

```
A o B o C
```

para expresar cualquiera de las condiciones

```
(A o B) o C
A o (B o C)
```

y similarmente para el operador y) y conmutativos (esto es, *A* **o** *B* y *B* **o** *A* son equivalentes, y similarmente para **y**), pero no pueden mezclarse directamente: aunque la Matematica define una prioridad entre estos operadores, poca gente la conoce, y cuesta saber si

```
A y B o C    (1)
```

significa

```
A y (B o C)    (2)
  o
A y B o C      (3)
```

Así, la expresión (1) no será correcta en este lenguaje, aunque sí lo serán las (2) y (3), distintas entre sí, que explicitan su significado mediante el uso de parentesis.

Operadores relacionales. "Conjuntos de caracteres"

Los *operadores relacionales*, de los que ya se ha visto un ejemplo (el operador "="), permiten comparar dos valores del mismo tipo. Definiremos los siguientes:

$A = B$	significa	*A es igual a B*
$A \neq B$	significa	*A no es igual a B*
$A \leq B$	significa	*A es menor o igual que B*
$A \geq B$	significa	*A es mayor o igual que B*
$A < B$	significa	*A es menor que B*
$A > B$	significa	*A es mayor que B*

Figura 9. Operadores relacionales

Su interpretacion, aplicada a valores de tipo *entero*, es la intuitiva; en el caso de los caracteres, por el contrario, no hay ninguna ordenacion "natural" de la que dispongamos para determinar, por ejemplo, si *'A'* es menor o no que *'9'*.

Podemos ver que el conjunto de valores del tipo *caracter* contiene varios subconjuntos heterogeneos entre si, algunos "naturalmente ordenados", como las letras y los numeros, y otros no, como los caracteres especiales. De todos modos, subsisten algunas preguntas, p. ej., "que sentido tiene decir que *'a'* es mayor, menor o igual que *'A'*?". Aunque la estructura natural de este conjunto seria el de un orden parcial, normalmente cada ordenador define un orden total sobre los caracteres mas o menos arbitrario, (añadiendo estructura a su conjunto: esto quiere decir que los ordena todos: podra compararse cualquier par de caracteres), que suele conocerse como *conjunto de caracteres* [del ingles "character set"]. Estos conjuntos de caracteres suelen variar entre ordenadores, y aun entre distintos modelos del mismo fabricante: algunos son mas amplios que otros (por ejemplo, los hay que no incorporan las letras minusculas). Las suposiciones basicas que cabe esperar que satisfaga todo conjunto de caracteres son las siguientes:

1. El espacio en blanco (*' '*) es *menor* que cualquier otro caracter "imprimible".[1]
2. Las letras estan *bien ordenadas* entre si, esto es, *'a' < 'b' < 'c' < ... < 'z'*, y *'A' < 'B' < 'C' < ... < 'Z'*.[2]
3. Los numeros tambien estan bien ordenados (*'0' < '1'*, etc.), y ademas son *contiguos*: no existe ningun caracter entre el *'0'* y el *'1'*, ni entre el *'1'* y el *'2'*, etc. La propiedad de contiguidad, deseable tambien para las letras, no se cumple para los conjuntos de caracteres de muchos ordenadores.

Estas propiedades permiten garantizar un funcionamiento correcto de la mayoria de programas que realizan tratamientos de caracteres.

Ejemplo 1: Contar los digrafos LA

Definicion del problema: Escribir un programa que analice una frase acabada por un punto y cuente el numero de *digrafos* (esto es, de secuencias de dos letras consecutivas) que sean *'LA'* (o *'la'*, *'La'*, *'lA'*). Supondremos que la frase contiene al menos una letra.

Analisis de una posible solucion: El problema es similar en su estructura al resuelto en el Algoritmo 3 en la pagina 10. Podremos, pues, utilizar el mismo esquema, aunque en este caso la unidad de tratamiento no sera el caracter, sino el par de caracteres o *digrafo*.

---

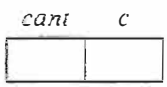
1 La mayoria de *conjuntos de caracteres* constan de un numero de caracteres "imprimibles" y de otros, llamados *caracteres de control*, que no lo son y suelen utilizarse para funciones internas de los dispositivos de representacion de textos, como saltos de pagina o cambios de linea.

2 En el alfabeto EBCDIC, utilizado por IBM, la propiedad de buena ordenacion no se cumple en el caso de las letras 'Ñ' y 'ñ'.

Diseño de un algoritmo: Puesto que solo hemos definido dos tipos (*caracter* y *entero*), no conocemos ningun metodo para definir una variable que contenga un par de caracteres, asi que nos vemos forzados a representar cada digrafo mediante dos variables

```
var cant, c: caracter;
```

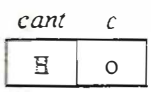
donde *c* representa el ultimo caracter que analizamos y *cant* el caracter anterior a este; y la pareja (*cant*,*c*), el par de caracteres que estamos analizando:



El paso de un par al siguiente, que corresponde a la operacion *lee c* del Algoritmo 3, sera aqui mas complejo. Hay que tener en cuenta que analizamos *todos* los pares de caracteres que contiene la frase:

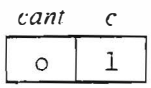
Si la frase es: `'Hola.'`, las parejas seran `'Ho'`, `'ol'`, `'la'` y `'a.'`.

Puede observarse que el segundo caracter de cada par pasa a ser el primer caracter del siguiente; en nuestros terminos, *cant* pasa a valer *c*; el caracter realmente nuevo es el segundo, y debera ser obtenido mediante *lee c*.



Antes de cambiar de pareja

al pasar al siguiente par de caracteres,



Despues de cambiar de pareja

cant toma por valor el anterior valor de *c*
c es el caracter recien leido

Asi, escribiremos

```
cant ← c; lee c;
```

para realizar el paso de un par al siguiente.

La condicion para incrementar la cuenta de digrafos sera que *cant* valga *'L'* (o *'l'*) y *'c'* valga *'A'* (o *'a'*):

```
(cant = 'L' o cant = 'l') y (c = 'A' o c = 'a')
```

Por ultimo, hay que tener en cuenta que, si deseamos realizar un tratamiento sistematico de todos los casos posibles, hemos de prever la posibilidad de que solo haya una letra en la frase. Esto puede complicar el programa (habria que mirar los digrafos solo si la frase tuviera mas de una letra), de modo que propondremos una solucion alternativa: considerar que la frase esta "extendida" con un espacio en blanco a la izquierda (puede verse facilmente que esta transformacion no altera el resultado del programa). Para ello, inicializaremos `c ← ' '` con lo que obtendremos la siguiente version del programa.

```
programa Cuenta_digrafos_LA es
  var cant,c: caracter; n: entero;
haz
  escribe_linea "Escribe una frase terminada por un punto:";
  c ← ' '; n ← 0;
  repite
    cant ← c; lee c;
    si (cant = 'L' o cant = 'l') y (c = 'A' o c = 'a') entonces
      n ← n + 1;
    fin si;
  hastaque c = '.';
  escribe_linea "El numero de digrafos LA que contiene esta frase es:",n,".";
fin programa;
```

Algoritmo 4. Contar los digrafos LA.

Ejemplo 2: Contar los digrafos LA interiores

Una modificacion del programa anterior consiste en contar solo los digrafos LA *que se encuentren a mitad de palabra*, esto es, no los que empiezan o terminan una. En este caso, habra que inspeccionar cuaternas (o cuadruplas; o grupos de cuatro) de caracteres: los situados en el medio habran de ser LA, y, mediante los extremos, controlaremos si nos hallamos o a mitad de palabra. Llamaremos *c3*, *c2*, *c1* y *c*, en este caso, a las variables, con la convencion expresada en la siguiente figura:

c3	*c2*	*c1*	*c*
(1)	L,l	A,a	(2)

(1) No debe ser un espacio en blanco
(2) No debe ser ni un espacio en blanco ni un punto

Para saber si no estamos a principio de palabra bastara con asegurarnos de que *c3* sea un espacio en blanco. Saber si estamos a final de palabra requiere, ademas de verificar que *c* no sea un blanco, verificar que no sea un punto, ya que podria ser que la ultima palabra de la frase terminase con el digrafo LA seguido inmediatamente de un punto; y, en este caso, no deberia ser contado.

El paso de una cuaterna a la siguiente seguira el mismo esquema que en el Algoritmo 4, aunque sera mucho mas laborioso. Asimismo, deberemos suponer que la frase se encuentra extendida a la izquierda con tres espacios en blanco.

Ofrecemos una version del programa completo:

```
programa Cuenta_digrafos_LA es
  var c3, c2, c1,c: caracter; n: entero;
haz
  escribe_linea "Escribe una frase terminada por un punto:";
  c ← ' '; c1 ← ' '; c2 ← ' '; n ← 0;
  repite
    c3 ← c2; c2 ← c1; c1 ← c; lee c;
    si no (c3 = ' ') y no (c = ' ' o c = '.') y
      (c2 = 'L' o c2 = 'l') y (c1 = 'A' o c1 = 'a') entonces
      n ← n + 1;
    fin si;
  hastaque c = '.';
  escribe_linea "El numero de digrafos LA interiores que contiene esta frase es:",n,".";
fin programa;
```

Algoritmo 5. Contar los digrafos LA interiores.

Se notara que las (sub)expresiones negadas, como **no** *(c3=' ')*, hubiesen podido ser escritas mas sencillamente como *c3≠' '*, utilizando el operador de desigualdad "≠". Se ha utilizado **no** para mostrar su funcionamiento.

Ejercicios

1. Mostrar las siguientes equivalencias (Leyes de Morgan):

   no (*A* y *B*) = no *A* o no *B*
   no (*A* o *B*) = no *A* y no *B*

2. Simplificar las siguientes expresiones logicas:

   no no *A*
   no (*A* y *B* y *C*)
   (*A* y *B*) o (*A* y no *B*)
   (*A* o *B*) y (*A* o no *B*)

3. Ver que cambios son necesarios en el Algoritmo 3 en la pagina 10 para que cuente las As mayusculas y minusculas.

4. Escribir un programa que cuente las apariciones de palabras terminadas en 'CION' (en una frase no vacia terminada por un punto).

5. Escribir un programa que, a partir de una lista de numeros terminada por un cero, cuente cuantos cambios de signo hay (esto es, cuantos pares de numeros *n* y *m* existen tales que *n* sea negativo y *m* positivo, o al reves), y cuantos pares de numeros coincidentes hay. Puede ser util recordar que *m* y *n* tienen distinto signo si y solo si $m*n < 0$.

Sintaxis y estilo

Definiremos un formalismo que nos permitira describir con total precision la sintaxis de las construcciones del lenguaje UBL. Esto es necesario (ya que las descripciones informales, utilizadas hasta ahora, son inadecuadas para describir con precision lenguajes de esta naturaleza), y desde luego conveniente (nos permite disponer de un criterio fiable para decidir sin ambiguedad cuales son las construcciones correctas en el lenguaje). Igualmente, definiremos convenciones estilisticas (diferentes de la sintaxis) que, sin ser obligatorias, permitiran, de ser seguidas, el intercambio y la comprension faciles de los programas.

Metalenguaje y metasimbolos. Reglas sintacticas

Todo lenguaje define reglas para la formacion de sus frases: un ejemplo de ellas es decir que, en castellano, una frase se compone de sujeto y predicado, y que el predicado se compone de verbo y complemento (aunque esto no es del todo cierto, y este tipo de analisis hace tiempo que no se aplica, vamos a suponerlo asi para simplificar la argumentacion). No hay ninguna dificultad en aceptar que esto puede expresarse mediante la siguiente formalizacion:

Frase = Sujeto Predicado
Predicado = Verbo Complemento

Estrictamente, las dos lineas anteriores constituyen de por si "frases", no en castellano, sino en el lenguaje utilizado para describir el castellano (que llamaremos *metalenguaje*): un metalenguaje es un lenguaje empleado para describir otro lenguaje. Llamaremos *reglas sintacticas* a las frases del metalenguaje.

El simbolo "=" es un ejemplo de lo que llamaremos *metasimbolo* (o simbolo del metalenguaje).

Supongamos que tratamos un subconjunto restringido del castellano en el cual las unicas palabras validas como sujeto son "Yo" y "tu". Conviniendo en que el metasimbolo "|" significa "o", escribiremos esto como

Sujeto = Yo | Tu

Sin embargo, caemos aqui en cierta confusion, ya que las palabras "Sujeto" y "Yo" o "Tu" pertenecen a categorias distintas. En efecto, pues "Sujeto" representa a cualquier sujeto, es una construccion generica perteneciente al metalenguaje, y en cambio "Yo" o "Tu" son palabras concretas del lenguaje. Llamaremos *no terminales* a las construcciones genericas, y *terminales* a los elementos del lenguaje. Utilizaremos algun tipo de distincion grafica (que se especificara mas adelante) para determinar si un simbolo es o no un metasimbolo: por ejemplo, podriamos escribir la regla anterior de alguna de las siguientes formas:

Sujeto = *Yo | Tu*
Sujeto = "Yo" | "Tu"

Un metalenguaje para la descripcion sintactica de UBL

Describiremos formalmente la sintaxis del lenguaje UBL mediante las siguientes convenciones:

1. Utilizaremos los simbolos =, (,), [,], {, } , | y % como metasimbolos; como tambien forman parte del lenguaje UBL, cuando sea necesario utilizarlos en este ultimo sentido los escribiremos entre comillas (comillas que no habran de reproducirse al escribir un programa):

   Signo_de_igualdad = "="

2. Distinguiremos los no terminales (esto es, las construcciones genericas) escribiendolas en minusculas (aunque es posible que la primera letra sea mayuscula), y los terminales que sean identificadores (lo que llamaremos mas adelante identificadores reservados y predefinidos), escribiendolos en negrita o mayusculas.

3. Las reglas sintacticas tendran siempre la forma

   No_terminal = Formula

   El metasimbolo = podra leerse "se escribe" o "se desarrolla".

4. Si una construccion X consiste en la construccion Y seguida de la construccion Z, escribiremos

   X = Y Z

   Se han visto ya varios ejemplos de esta operacion.

5. Utilizaremos el metasimbolo | como alternativa, y lo leeremos "o": con lo que se ha explicado en capitulos anteriores, podemos escribir

   Declaracion_de_objeto = Declaracion_de_variable | Declaracion_de_constante

6. Encerraremos una construccion entre los metasimbolos [y] cuando, en el desarrollo de determinada formula, pueda prescindirse de esa construccion. Por ejemplo, la sintaxis de la sentencia **si** podria ser:

   si *condicion* **entonces**
   instruccion(es)
   [**sino**
   instruccion(es)]
   fin [**si**];

   Indicando con los corchetes la posibilidad de prescindir, en determinada instruccion **si**, de la parte precedida por **sino** y/o del **si** final.

7. Encerraremos una construccion entre los metasimbolos { y } cuando, en el desarrollo de determinada formula, pueda escribirse un numero arbitrario (incluido cero) de veces esa construccion. Por ejemplo,

   {*instruccion*}

   representara una secuencia de cero o mas instrucciones; y

   instruccion {*instruccion*}

   representara una o mas instrucciones, con lo cual podria escribirse

   instruccion(es) = instruccion {*instruccion*}

   para dar un sentido mas formal a la construccion "instruccion(es)" que ha aparecido ya en algunas formulas.

8. Utilizaremos el metasimbolo % para encerrar meta-comentarios, descripciones en castellano que substituiran a formalizaciones mas estrictas en casos en los que sea imprescindible (un ejemplo de utilizacion de "%" se encuentra a continuacion).

9. Por ultimo, utilizaremos los metasimbolos (y) como meta-parentesis para evitar ambiguedades en la interpretacion de las formulas.

Un ejemplo: Alfabeto del lenguaje UBL.

Como ejemplo real de aplicacion del formalismo definido, damos la descripcion del alfabeto basico del lenguaje UBL: contiene los caracteres que toda version del lenguaje debe proporcionar (aunque cada version pueda definir caracteres adicionales). Los caracteres estan agrupados por categorias.

```
caracter =
  letra | digito | caracter_especial | otros_caracteres | %espacio en blanco%

letra =
  A|B|C|D|E|F|G|H|I|J|K|L|M|N|Ñ|O|P|Q|R|S|T|U|V|W|X|Y|Z|
  a|b|c|d|e|f|g|h|i|j|k|l|m|n|ñ|o|p|q|r|s|t|u|v|w|x|y|z

digito = 0|1|2|3|4|5|6|7|8|9

caracter_especial =
  "("|")"|"["|"]"|"{"|"}"|+|-|*|/|"|"|"!"|'|↑|_|<|>|:|;|.|,|=

otros_caracteres = %otros caracteres, segun la version%
```

Figura 10. *Alfabeto del lenguaje UBL*

Simbolos

El texto que forma un programa se compone de *simbolos* (como "n", "**var**", "+" o "←"), que a su vez se componen de caracteres del alfabeto. El conjunto de los simbolos posibles en UBL constituye su *vocabulario.*

```
identificador = letra{[_](letra|digito)}

tira = "{caracter}"

caracter_constante = 'caracter'

numero_sin_signo = entero_sin_signo | real_sin_signo

entero_sin_signo = digito{digito}

real_sin_signo =
  (parte_entera.parte_decimal[exponente]) | (parte_entera[.parte_decimal]exponente)

  parte_entera = entero_sin_signo

  parte_decimal = entero_sin_signo

  exponente = (E|e)[+|-]entero_sin_signo

simbolo_especial = caracter_especial | ← | ≤ | || | ≠ | ≥ | => | ..
```

Figura 11. *Vocabulario del lenguaje UBL:* Notese la introduccion de no terminales auxiliares en la descripcion de la sintaxis de un *real_sin_signo.*

Identificadores; identificadores reservados y predefinidos: Los identificadores representan abstracciones y se asocian a distintos tipos de entidades. Deben comenzar con una letra, y pueden seguir con numeros y/o letras -- no se permiten blancos intermedios, ni caracteres especiales. Puede escribirse el caracter de subrayado entre dos caracteres (pero no dos subrayados seguidos, ni uno al final), para aumentar su legibilidad. Si dos identificadores difieren solo en que alguna letra es mayuscula en uno y minuscula en otro, se consideran iguales.

Asi,

A X Universidad_de_Barcelona Ceñudo X25 X_1_1_1

son identificadores validos;

resultado Resultado RESULTADO ReSuLtAdO

son cuatro maneras de escribir el mismo identificador; y

Universidad de Barcelona
A__Y__B
1X2
*A_*
Mas_o_-

no son [validos como] identificadores [el primero contiene blancos, el segundo dos subrayados seguidos, el tercero no empieza por una letra, el cuarto termina con un subrayado, y el quinto contiene caracteres especiales].

Algunos identificadores tienen un significado especial dentro del lenguaje UBL; a diferencia de los demas, no pueden utilizarse libremente asociandolos con abstracciones, sino que cumplen determinado papel en la sintaxis y semantica de un programa. Se les llama *identificadores reservados* o *palabras clave*; en este manual se representaran siempre en negrita.

Asimismo, determinadas abstracciones (como los tipos *entero* y *caracter* o las acciones *lee* y *escribe*) estan definidas automaticamente. Llamaremos *identificadores predefinidos* a los nombres que las representan. Al final del capitulo se encuentran tablas de identificadores reservados y predefinidos en UBL Catalan y Castellano.

Nota: Mientras que las palabras clave no pueden utilizarse como identificadores en un programa (y por eso se llaman *reservadas*), no hay ningun problema en utilizar identificadores iguales a los predefinidos para representar abstracciones en un programa, sabiendo que la abstraccion predefinida representada por el identificador no podra utilizarse si se redeclara (esto se comprendera mas claramente mas adelante, al hablar de declaraciones y reglas de reconocimiento de nombres).

Caracteres y tiras de caracteres: Las tiras (de caracteres) representan informacion alfanumerica constante de cualquier longitud (numero de caracteres) y se "encierran" entre comillas. Pueden contener caracteres del alfabeto basico (incluido el espacio en blanco) y otros que posea el ordenador que se utilice. Para evitar ambiguedades, si alguno de los caracteres de la tira es una comilla ("), esta se escribira dos veces: asi, 'Dijo: "Callate"' se escribira

```
"Dijo: ""Callate""".
```

Una tira tambien puede no contener ningun caracter (en cuyo caso se representa ""); la llamaremos *tira vacia*. Una tira debe estar siempre contenida en una linea; es decir, no son posibles construcciones del tipo

```
"Esta tira de caracteres no es
valida en el lenguaje UBL"
```

Un caracter constante es un caracter encerrado entre apostrofes y representa informacion alfanumerica de longitud 1. Se aplican las mismas observaciones que a las tiras; en particular, el caracter "'" se escribira ''''.

Numeros: El lenguaje UBL proporciona dos tipos de datos (ademas de los definibles por el programador) para tratar informacion numerica: el tipo *entero* (que ya se ha estudiado), y el tipo *real* (que se vera mas adelante).

Los enteros sin signo representan numeros enteros no negativos; cada version de UBL puede establecer un maximo y un minimo en el conjunto de enteros aceptables. Son ejemplos de "entero_sin_signo"

1
23
77
12345678

Los reales sin signo representan numeros reales, escritos en notacion exponencial (la E se lee "por diez elevado a"). Se distinguen de los enteros en que se escriben con punto decimal y/o exponente. Notese que, de escribirse el punto decimal, este debe ir seguido de al menos un digito. Cada version puede imponer limites en cuanto al numero de digitos significativos y la magnitud del exponente.

Son *reales_sin_signo*

```
12
3.14159265535
8.11111E-23
23E-23
```

pero no

```
1.
.23
E10
```

Simbolos especiales Los simbolos especiales incluyen separadores ("..","."."."."."."|"), utilizados como "signos de puntuacion" del lenguaje, y operadores (como "÷" o "-").

Comentarios

En ocasiones puede ser necesario tener informacion relativa a un programa (explicacion del algoritmo utilizado, autor y fecha del programa, que representa cada variable). Esto puede hacerse mediante algun mecanismo de documentacion externa (creando un fichero que contenga esa informacion) o mediante el uso de *comentarios* intercalados en el programa. Un comentario es un fragmento de texto que no forma parte del sentido del programa, pero completa la informacion que proporciona. Por ejemplo, una declaracion como

```
var n: entero;
```

puede ser mas clara si se escribe como

```
var n: entero; -- cuenta las As
```

"cuenta las As" es un comentario.

El lenguaje UBL proporciona dos modos de escribir comentarios: precedido por los caracteres "--", cualquier texto que no ocupe mas de una linea, como en el ejemplo anterior; o, encerrado entre "(*" y "*)", cualquier texto (ocupando cualquier numero de lineas):

```
var n: entero; (* cuenta las As *)
```

Los comentarios son ignorados por el compilador (aunque aparecen en el listado); su unico sentido es el de documentar un programa.

Dos errores frecuentes al utilizar comentarios son la *subdocumentacion* y la *sobredocumentacion*: un programa (especialmente si es largo) pobremente documentado sera dificilmente comprensible; y demasiados comentarios en un programa oscureceran la comprension de este: debe procurarse, mediante la eleccion de identificadores mnemotecnicos y la distribucion adecuada del texto, eliminar la documentacion superflua.

Un comentario puede encontrarse dentro de una tira de caracteres; en tal caso, se considera como grupo de caracteres y no como comentario. La ambiguedad producida al considerar si

```
" Esta tira es (* ambigua *) "
```

significa

```
" Esta tira es "
```

(tomando "(* ambigua *)" como un comentario), o bien significa lo que

```
" Esta tira es (* ambigua *) "
```

se resuelve mediante esta regla.

Escritura de programas

Los espacios en blanco no tienen significado alguno para el compilador, y pueden omitirse, excepto en dos casos:

- En las tiras de caracteres y los caracteres literales, donde cada espacio cuenta como un caracter; y
- Entre dos simbolos que, de escribirse juntos, formarian otro simbolo: por ejemplo, "**si** *A*" no es equivalente a "si*A*", que se intrepretara como el identificador *siA* y no como la palabra reservada **si** seguida del identificador *A*.

Ademas, el final de una linea se intrepreta como un espacio en blanco adicional, lo cual impide "partir" un simbolo entre en final de una linea y el principio de otra.

Asi, el Algoritmo 3 en la pagina 10 podria reescribirse

```
programa Cuenta_las_As es var c:caracter;n:entero;haz escribe_linea
" Escribe una frase terminada por un punto ";n←0;repite lee c;si c='A'entonces
n←n+1;fin;hastaque c='.';escribe_linea
" El numero de letras A que hay en esta frase es ",n;fin programa;
```

Algoritmo 6. Ejemplo de programa ilegible

aunque no recomendamos este estilo.

Convenciones de estilo

Un programa como el mostrado en el anterior ejemplo es muy dificilmente comprensible (si lo es en absoluto), aparte de ser esteticamente horrible. Para evitar en lo posible la proliferacion de tales programas, sugeriremos diversos estilos de escritura para cada una de las construcciones del lenguaje; aunque estas convenciones se explicaran con la sintaxis, debe entenderse que no son obligatorias, en el sentido de que no afectan a la posibilidad de proceso de un programa por un interprete automatico, sino unicamente destinadas a facilitar la legibilidad, mantenimiento e intercambiabilidad de los algoritmos.

- Llamaremos *indentacion* de una linea en un programa al numero (eventualmente nulo) de espacios en blanco que la preceden (este concepto es conocido en castellano bajo el nombre de *sangrado*). Igualmente, diremos que una linea esta "indentada tres espacios", o bien que su *nivel de indentacion* es tres. El uso cuidadoso de la indentacion sera la base de nuestras convenciones estilisticas.

  Existe una subordinacion logica entre las instrucciones (y declaraciones; como se vera) de un programa: si una instruccion *I* debe repetirse hasta la verificacion de una condicion *C*,

  ```
  repite
    I
  hastaque C;
  ```

  podemos decir que *I* es una instruccion subordinada (o sub-instruccion) de la instruccion **repite** completa. Expresaremos esto mediante la indentacion de la instruccion *I* un numero fijo de espaci+os (en este manual utilizamos dos). Igualmente, si la repeticion debiera ejecutarse solo en el caso del cumplimiento de una condicion *D*, escribiriamos

  ```
  si D entonces
    repite
      I
    hastaque C;
  fin si;
  ```

- En el caso de que una instruccion no compuesta o una declaracion ocupe mas de una linea, las lineas adicionales se distinguiran mediante una indentacion adicional. Ejemplos:

```
n ← a ^ b * c − a * b * d − a * c * b − b * c * d +
  2 * a * b * c * d.
escribe "El valor de N es: ",n,", el de M es ",m,
  ", y el de K es ",k.
```

Presentamos a continuacion la sintaxis y los modelos de indentacion sugeridos para las construcciones explicadas; las variantes propuestas abarcan todos los casos posibles.

Sintaxis

si *condicion* **entonces**
  instruccion
  {*instruccion*}
[**sino**
  instruccion
  {*instruccion*}]
fin [**si**];

Estilo

Caso general:

```
si condicion entonces
  instruccion(es)
sino
  instruccion(es)
fin si;
```

En caso de que falte ***sino***, se escribira:

```
si condicion entonces
  instruccion(es)
fin si;
```

Si las instruccion(es) caben en una sola linea, puede utilizarse

```
si condicion entonces instruccion(es)
sino instruccion(es)
fin si;
```

Omitiendo la parte **sino** si procede, o utilizando formas mixtas con las anteriores.

Si la instruccion ***si*** entera cabe en una sola linea,
puede utilizarse

```
si condicion entonces instruccion sino instruccion fin;
```

Observese en este caso la ausencia del **si** final (no obligatoria).

Figura 12. Sintaxis y estilos de escritura para la instruccion **si**

Los estilos mas compactos se utilizaran solo si se desea; siempre es posible utilizar el descrito en el caso general. Lo mismo se aplica a las demas convenciones estilisticas; que, por otra parte, se describen mas informalmente que la sintaxis.

---

variable ← *expresion*;

Figura 13. Sintaxis de la instruccion de asignacion

Sintaxis

repite
  instruccion
  {instruccion}
hastaque *condicion;*

Estilo

Caso general:

repite
  instruccion(es)
hastaque *condicion;*

Si la repeticion entera cabe en una sola linea:

repite *instruccion* **hastaque** *condicion;*

Figura 14. ***Sintaxis y estilos de escritura para la instruccion*** **repite**

---

programa *identificador* **es**
  declaracion
  {declaracion}
haz
  instruccion
  {instruccion}
fin programa;

Figura 15. ***Sintaxis y estilo de escritura para un*** **programa**

Sintaxis

*[***var***] identificador {,identificador}: tipo;*
const *identificador = expresion_constante;*

Estilo

var *identificador(es): tipo; {identificador(es): tipo}*
const *identificador = expresion_constante;*

Figura 16. ***Sintaxis y estilos de escritura para las declaraciones de objeto:*** Podran juntarse varias declaraciones de variable en una sola linea utilizando una sola vez la palabra **var**. Las declaraciones de constantes se escribiran siempre cada una en una linea.

Ejercicios

1. Caracterizar las formaciones resultantes de la regla

   q = [1] [X] [2]

2. Sin utilizar la autoalusion, no es posible escribir una regla que describa el conjunto de las formaciones que consisten en un numero arbitrario de As seguido del mismo numero de Bs.

3. Los ceros a la izquierda en los numeros enteros no son significativos. Escribir un fragmento modificado de la sintaxis del lenguaje UBL que no permita los ceros a la izquierda. [**Indicacion:** dividir la categoria sintactica *digito* en dos, como *digito_no_nulo* y *cero*].

Apendice: tablas de identificadores reservados y predefinidos en UBL catalan y castellano.

acaba	entonces	modulo	sal
accion	es		secuencia
aplicacion	existe	nada	segun
		no	si
ciclico	fila	nombre	sino
con	fin	nulo	
condicion	funcion		tabla
conjunto		o	talque
const	hastaque	otros	tipo
cuando	haz		tupla
		para	usa
de	implementa	produce	
decide	itera	programa	vale
div			var
	mientras	repite	
en	mod		y

Figura 17. Tabla de identificadores reservados en UBL Castellano

acaba	es	mod	res
accio	existeix	modul	
altres			segons
amb	fes	no	sequencia
aplicacio	fi	nom	si
	fila	nul	sino
ciclic	finsque		surt
condicio	funcio	o	talque
conjunt			taula
const	i	per	tipus
	implementa	produeix	tupla
de	itera	programa	
decideix			usa
div	llavors	quan	val
en	mentre	repeteix	var

Figura 18. Tabla de identificadores reservados en UBL Catalan

abs	*desconecta*	*impar*	*pon*
asc	*entero*	*lee*	*pred*
caracter	*entrada*	*lee_linea*	*real*
cierto	*escribe*	*libera*	*salida*
conecta	*escribe_linea*	*logico*	*suc*
crea	*falso*	*long*	*texto*
cuadrado	*fdf*	*obten*	*tira*
desc	*fdl*	*ordinal*	*trunc*

Figura 19. Tabla de identificadores predefinidos en UBL Castellano.

abs	*enter*	*llegeix*	*pred*
asc	*entrada*	*llegeix_linia*	*quadrat*
caracter	*escriu*	*llibera*	*real*
cert	*escriu_linia*	*logic*	*sortida*
conecta	*fals*	*long*	*suc*
crea	*fdf*	*obte*	*texte*
desc	*fdl*	*ordinal*	*tira*
desconecta	*imparell*	*posa*	*trunc*

Figura 20. Tabla de identificadores predefinidos en UBL Catalan.

Diseño descendente. Acciones y Condiciones

Los conceptos introducidos en capitulos anteriores permiten el diseño de programas arbitrariamente complejos: sin embargo, no hemos presentado ningun metodo para su elaboracion, sino que hemos sugerido, mostrandola en la practica con varios ejemplos, la conveniencia de *analisis teoricos* y *expresiones en el lenguaje* de esos analisis. Describiremos detalladamente una aproximacion a la construccion sistematica de algoritmos que se conoce como *diseño descendente* (en ingles, *top-down design*) o *metodo de refinamiento progresivo*, asi como algunas partes del lenguaje UBL adecuadas a ese metodo.

Un ejemplo de diseño descendente

Disponemos de un texto formado por frases no vacias acabadas en un punto; el texto mismo se termina con un asterisco que sigue inmediatamente al ultimo punto. Deseamos escribir un programa que cuente el numero medio de As que aparecen en las frases del texto. Una primera aproximacion podria ser:

```
-- Numero Medio de As. Refinamiento 1 --
programa numero_medio_de_As es
haz
  presentate;
  repite
    cuenta_las_as_de_una_frase;
    acumula_cuenta_de_As;
    cuenta_la_frase;
  hastaque se_termine_el_texto;
  calcula_la_media;
  escribe_resultados;
fin programa;
```

Si el lenguaje UBL "entiendese" directamente las instrucciones y condiciones que hemos utilizado (p. ej., si formasen parte de su repertorio de entidades predefinidas), el programa estaria terminado. Este no es el caso, de modo que sera necesaria una mayor elaboracion de cada instruccion hasta hacerla comprensible por el interprete; el programa mostrado puede tomarse, de todos modos, como una primera version o esquema del algoritmo completo. La construccion del programa pasara ahora por un *refinamiento* o detalle de sus instrucciones y condiciones, que podran considerarse, hasta cierto punto, como (sub)algoritmos a desarrollar independientemente.

Cada desarrollo nos conducira a una nueva version del programa, que asi habremos expresado segun diferentes *niveles de abstraccion*, al variar el detalle y precision de sus instrucciones: la primera version sera la mas abstracta -- probablemente, tambien la mas comprensible para el lector y la que menos suposiciones no relativas al problema hace --, y las sucesivas ganaran en detalle y perderan en generalidad.

Encontramos aqui dos de las ideas fundamentales del diseño descendente: la de *refinamiento*, que permite "suponer el problema resuelto" y concentrarse en la estructura del programa, sin hacer referencia a un modelo predeterminado: y la de *division* del problema en subproblemas posiblemente mas sencillos e independientes.

Para terminar el programa, podemos adoptar varias estrategias; la mas sencilla sera aprovechar el esquema del Algoritmo 3 en la pagina 10 para refinar *cuenta_las_As_de_una_frase*. Esto nos obligara a declarar variables c y n como en ese programa; ademas, necesitaremos variables para acumular el numero total de As y el numero de frases; las llamaremos respectivamente t y f. El desarrollo del resto del programa es trivial, comparando tambien la estructura general con el esquema del Algoritmo 3:

```
-- Numero Medio de As. Refinamiento 2a --
programa numero_medio_de_As es
  var c: caracter; n, t, f: entero;
haz
  -- Inicializacion --
  escribe_linea "Escribe un texto terminado por un asterisco";
  t ← 0; f ← 0;
  -- Proceso del texto --
  repite
    -- proceso de una frase --
    n ← 0;
    repite
      lee c;
      si c = 'a' o c = 'A' entonces n ← n + 1; fin;
    hastaque c = '.';
    -- acumula resultados, pasa de frase --
    lee c;
    f ← f + 1;
    t ← t + n;
  hastaque c = '*';
  -- escribe resultados --
  escribe_linea "El numero medio de As por frase es: ",t div f;
fin programa;
```

Otra solucion al mismo problema. Instrucciones nada y decide

Una consideracion mas atenta del problema permite ver que solo se precisa contar el numero de As y el numero de puntos del texto (ya que lo suponemos bien escrito), y que para ello basta con una sola iteracion; cada una de ellas debera actuar de modo distinto segun el caracter examinado sea un punto, una A u otro caracter: se trata de una toma de decision con mas de dos alternativas, que podria escribirse utilizando combinaciones de instrucciones **si**:

```
si c = 'A' o c = 'a' entonces ...
sino
  si c = '.' entonces ...
  sino ...
  fin si;
fin si;
```

Este tipo de construccion, como se vera, es bastante frecuente, y resulta poco clara escrita de este modo. Introduciremos pues otra forma de toma de decision, la instruccion **decide**, mediante la cual lo anterior podra expresarse

```
decide
  cuando c = 'A' o c = 'a' ⇒ ...
  cuando c = '.' ⇒ ...
  cuando otros ⇒ ...
fin decide;
```

La forma general de la instruccion **decide** es

```
instruccion_decide =
  decide
    cuando condicion ⇒
      instruccion
      {instruccion}
    {cuando condicion ⇒
      instruccion
      {instruccion}}
    [cuando otros ⇒
      instruccion
      {instruccion}]
  fin [decide]:
```

Figura 21. Sintaxis y estilo de la instruccion **decide:** si la (o las) instruccion(es) asociadas a una alternativa caben a continuacion de la flecha ⇒), pueden escribirse juntas en esa linea.

Efecto de la ejecucion de la instruccion decide: se evaluan secuencialmente las condiciones escritas entre **cuando** y "⇒", en el orden en que estan escritas, hasta encontrar una que sea cierta, en cuyo caso se ejecutan las instrucciones asociadas. Si ninguna condicion es cierta, se ejecutan, si existen, las instrucciones asociadas a **cuando otros**; si se omite **cuando otros**, se produce un error si no se verifica ninguna de las condiciones. Una vez determinada la alternativa a ejecutar, no se evalua ninguna otra condicion.

Dado que se evaluan todas las condiciones hasta encontrar una que sea cierta, sera conveniente de cara a la eficiencia del programa escribir las condiciones en orden decreciente de probabilidad, con objeto de que la maxima evaluacion se produzca en el minimo numero de casos.

Definimos tambien una instruccion **nada**, de efecto nulo, que sera util entre otras cosas en el tratamiento de determinadas alternativas que no requieren ejecucion alguna:

```
instruccion_nada = nada;
```

Figura 22. Sintaxis de la instruccion **nada**

Volviendo al programa que nos ocupa, podemos ver que las tres instrucciones contenidas dentro de la iteracion del Refinamiento 1 se "funden", al elaborarlas, en una sola instruccion **decide**, y que la variable *N* del refinamiento 2a es innecesaria (ya que en este caso no utilizamos como base el Algoritmo 3). Esto es usual al utilizar el metodo de diseño descendente: es posible que un refinamiento obligue a replantear el esquema del cual proviene. El programa completo podra escribirse:

```
-- Numero Medio de As. Refinamiento 2b --
programa numero_medio_de_As es
  var c: caracter; t, f: entero;
haz
  -- Inicializacion --
  escribe_linea "Escribe un texto terminado por un asterisco";
  t ← 0; f ← 0;
  -- Proceso del texto --
  repite
    lee c;
    -- seleccion de caracteres --
    decide
      cuando c = 'A' o c = 'a' ⇒ t ← t + 1;
      cuando c = '.' ⇒ f ← f + 1;
      cuando otros ⇒ nada;
    fin decide;
  hastaque c = '*';
  -- escribe resultados --
  escribe_linea "El numero medio de As por frase es: ",t div f;
fin programa;
```

Algoritmo 7. Media de las As por frase en un texto.

Es interesante observar que esta version es mas corta y eficiente que la primera (2a), al haberse escrito pensando la solucion sin recurrir a resultados establecidos. Tambien se notara que este programa no requiere que el asterisco final siga inmediatamente al ultimo punto, asi como el orden en que se han escrito las condiciones, basado en la suposicion razonable de que cada frase contendra varias letras A. Se ha utilizado "**cuando otros ⇒ nada;**" para evitar errores de ejecucion al procesar caracteres distintos del punto y la A.

Acciones y Condiciones. Instrucciones vale y acaba

El metodo expuesto tiene el inconveniente de que la version final del algoritmo no refleja el proceso de diseño a que ha sido sometido, que en ocasiones interesa mas en el problema que el propio programa. Para evitar esta perdida de informacion, el lenguaje UBL proporciona dos mecanismos de declaracion de entidades, que llamaremos **acciones** y **condiciones**, para aumentar el universo lexico disponible mediante la introduccion de abstracciones que representan instrucciones y condiciones.

Para ceñirnos a un ejemplo, supongamos que queremos escribir un programa como el del refinamiento 2a, pero sin perder la informacion contenida en el 1. Podemos hacer esto "explicando" al interprete, mediante declaraciones apropiadas, cual es el significado de instrucciones como *presentate*:

```
accion presentate haz
  escribe_linea "Escribe un texto terminado por un asterisco";
  t ← 0; f ← 0;
fin presentate;
```

Esta declaracion introduce una nueva abstraccion que permite utilizar *presentate;* como una nueva instruccion (en este caso, definida por nosotros). Igualmente, podemos indicar el sentido de *se_termine_el_texto* mediante la declaracion

```
condicion se_termine_el_texto haz
  vale c = '*';
fin se_termine_el_texto;
```

Mientras que la declaracion de *presentate* es una instruccion abstracta, *se_termine_el_texto* es una abstraccion de evaluacion: puede contener instrucciones como en la **accion** anterior, pero debe incluir (y ejecutar) al menos una instruccion **vale** (que expresa cual es el valor asociado a la abstraccion que representa y termina la ejecucion de esta).

---

instruccion_vale = **vale** *condicion;*

Figura 23. Sintaxis de la instruccion vale en condiciones

Utilizando este metodo, obtenemos el siguiente programa:

```
-- Numero Medio de As. Refinamiento 2c --
programa numero_medio_de_As es

  var c: caracter; n, t, f: entero;

  accion presentate haz
    escribe_linea "Escribe un texto terminado por un asterisco";
    t ← 0; f ← 0;
  fin presentate;

  accion cuenta_las_As_de_una_frase haz
    n ← 0;
    repite
      lee c;
      si c = 'A' o c = 'a' entonces n ← n + 1; fin;
    hastaque c = '.';
  fin cuenta_las_As_de_una_frase;

  accion acumula_cuenta_de_As haz t ← t + n; fin;

  accion cuenta_la_frase haz lee c; f ← f + 1; fin;

  condicion se_termine_el_texto haz vale c = '*'; fin;

  accion calcula_la_media haz nada; fin;

  accion escribe_resultados haz
    escribe_linea "El numero medio de As por frase es: ",t div f;
  fin escribe_resultados;

haz
  presentate;
  repite
    cuenta_las_as_de_una_frase;
    acumula_cuenta_de_As;
    cuenta_la_frase;
  hastaque se_termine_el_texto;
  calcula_la_media;
  escribe_resultados;
fin programa;
```

Algoritmo 8. Contar la media de As por frase en un texto (con acciones y condiciones)

El programa es en este caso mucho mas largo, pero puede argumentarse que es tambien mas comprensible. La efectividad de la notacion quedara clara, de todos modos, en programas mas complejos.

Exponemos ahora los mecanismos generales de declaracion de acciones y condiciones, asi como las instrucciones asociadas con su utilizacion.

```
declaracion_de_accion =
  accion identificador haz
    instruccion
    {instruccion}
  fin [identificador];
```

Estilo: ademas del sugerido en la sintaxis, puede utilizarse:

```
accion identificador haz instruccion {instruccion} fin;
```

Figura 24. Sintaxis y estilo para declaraciones de acciones *(Simplificado)*

Una declaracion de este estilo establece una asociacion entre el identificador y las instrucciones, de modo que la aparicion del identificador en el programa indica la ejecucion de las instrucciones asociadas. Llamaremos a tal instruccion *invocacion* a una accion:

instruccion_de_invocacion = *identificador*.

Figura 25. Sintaxis de la instruccion de invocacion (Simplificada)

Las condiciones se declaran de modo similar:

declaracion_de_condicion =
condicion *identificador* **haz**
instruccion
{*instruccion*}
fin [*identificador*];

Estilo: ademas del sugerido en la sintaxis, puede utilizarse:

condicion *identificador* **haz** *instruccion* {*instruccion*} **fin**;

Figura 26. Sintaxis y estilo para declaraciones de **condicion** *(Simplificado)*

En este caso se asocia el identificador con la ejecucion de las instrucciones asociadas, que debe terminarse con la de una accion **vale**: esta proporciona el valor de la condicion, que puede utilizarse tambien en expresiones mas complejas. En cuanto a las acciones, puede utilizarse la instruccion **acaba** para terminar incondicionalmente la ejecucion del subprograma:

instruccion_acaba = **acaba**.

Figura 27. Sintaxis de la instruccion **acaba**

Resumiendo: las **acciones** y **condiciones** declaran abstracciones (de instruccion y evaluacion, respectivamente). La aparicion en el programa del identificador que las representa tiene como efecto la *activacion* del respectivo *subprograma* (que tambien llamaremos *bloque*) y la ejecucion de las instrucciones asociadas. Durante la ejecucion de esas instrucciones, diremos que el bloque esta *activo*; dejara de estarlo al *terminar* esa ajecucion. cosa que puede suceder naturalmente (en el caso de una **accion**) o como efecto de la ejecucion de las instrucciones **acaba** y **vale**.

Notas

Al refinar un subprograma, pueden introducirse nuevos subprogramas; no hay problema en ello, mientras se respeten las siguientes reglas:

- No puede haber *colision de nombres*; de otro modo, no pueden declararse dos entidades con el mismo identificador.
- Toda entidad debe declararse *antes* de ser utilizada.

Ejercicios

1. Reescribir el Algoritmo 3 en la pagina 10 utilizando la tecnica de diseño descendente en los dos modos mostrados en este capitulo. Hacer lo mismo con los demas programas realizados hasta el momento.

2. El Maximo Comun Divisor (*mcd*) de dos numeros positivos es el mayor numero que los divide exactamente a ambos. Diseñar un algoritmo que calcule el mcd de dos numeros positivos distintos, utilizando tan solo las propiedades

   mcd(x,y) = mcd(x-y,y) -- suponiendo x > y
   mcd(x,y) = mcd(y,x)
   mcd(x,x) = x

3. Dada un digrafo y un texto, calcular la media de apariciones de ese digrafo por frase.

4. Dada una serie ascendente de numeros terminada por un cero, encontrar la longitud de la mayor subserie tal que cada miembro divide al siguiente.

Tiras de Caracteres

Muchas veces interesa procesar informacion alfanumerica a nivel complejo: en aplicaciones de tratamiento de textos, puede necesitarse trabajar con palabras o frases, tal como se utilizan los enteros en aplicaciones numericas o los caracteres en proceso elemental de textos. Para ello se introduce el tipo *tira*, del cual son literales, como ya se dijo, los simbolos definidos como "*tira*" en el vocabulario y escritos entre comillas.

A diferencia de los enteros y caracteres, que en cierto sentido son "atomicos", las tiras se componen de (cero o mas) caracteres individuales: diremos que son un ejemplo de *tipo estructurado*. Toda tira se compone de un determinado numero de caracteres, que llamaremos *longitud* de la tira. Asi,

"Esta es una tira de caracteres"
"Esta tambien; su longitud es 31"
""
"La anterior es una Tira Vacia"
"Toda comilla interior ("") debe duplicarse"

son tiras de caracteres (recordamos que se conoce como *tira vacia* a la tira de longitud 0 denotada por "").

Declararemos una variable de tipo tira del siguiente modo:

var *v : tira(10);*

indicando con el numero entre parentesis la *longitud maxima* de los valores que puede tomar la variable:

Podra hacerse
  v ← *"Barcelona";*
o
  v ← *"";*
pero no
  v ← *"Esta tira tiene mas de 10 caracteres";*
ya que hemos declarado un maximo de diez caracteres para la variable v.

Hablaremos tambien de la *longitud actual* de una variable de tipo tira, indicando la longitud de su valor, para distinguirla de la longitud maxima o declarada. Asi, podemos formular la siguiente

Definicion: Llamaremos *error de truncacion* al que ocurre al intentar asignar a un objeto de tipo tira un valor cuya longitud actual es mayor que su longitud maxima.

En este sentido, las variables de tipo tira son *ajustables*.

Convendremos en que todos los objetos de tipo tira son *compatibles*, en el sentido de que son asignables entre si (aunque pueden no tener el mismo tipo: tira(4) y tira(10) son tipos distintos compatibles). Los objetos constantes se declaran de la forma usual:

const *titulo = "Manual del Usuario del Lenguaje UBL";*

Operaciones sobre tiras

Ademas de la asignacion, distinguiremos las siguientes operaciones basicas sobre tiras (de las cuales pueden deducirse las demas):

- La *concatenacion* de tiras, denotada mediante el operador "||", que produce como resultado, a partir de dos tiras *T1* y *T2*, la tira formada por *T1* seguida de *T2*:

  "Barce" || "lona" = "Barcelona"

- La *seleccion* de un caracter, que permite obtener el valor de uno de los caracteres individuales que componen la tira: se expresa la posicion o *indice* del caracter deseado escribiendola entre corchetes despues del nombre de la variable:

  t ← "Anagrama"; -- t[1] = 'A'; t[4] = 'g'; t[8] = 'a'; t[9] produce un error

  Es un error si se intenta acceder a una posicion de la tira inexistente (esto es, si el indice es menor que 1 o mayor que la longitud de la tira). El tipo del resultado es *caracter*.

- La *subtira*, que permite obtener un trozo de una tira, limitada entre dos indices:

  t ← "ABCDEFGH";
  -- t[1..3] = "ABC";
  -- t[3..2] = ""
  -- t[4..4] = "D" (distinto de T[4] = 'D')

  Los indices estan sometidos a las mismas restricciones que en el caso anterior; en este, el resultado es un valor de tipo *tira* (no *caracter*) con una longitud igual a la resta del segundo indice (o *limite superior*) y el primero (o *limite inferior*) menos 1. Si el limite superior es menor que el limite inferior, convendremos en que el resultado es la tira vacia.

- La *longitud* de una tira puede determinarse usando la funcion predefinida *long*:

  t ← "123"; -- long(t) = 3

- La *conversion a tira* de un caracter es el medio para "generar" tiras a partir de otros objetos: se escribe

  tira(c)

  donde *c* es un caracter, para denotar la tira de longitud 1 cuyo unico componente es el caracter *c*. [Esta funcion se incluye para respetar la regla de compatibilidad de tipos: puesto que todo objeto tiene un tipo, no seria posible asignar un caracter a una tira directamente].

Comparacion de tiras

La comparacion de objetos de tipo tira se efectua utilizando la *ordenacion lexicografica*, alfabetica o natural sobre las tiras e ignorando blancos a la derecha. Queremos decir que

1. Dadas dos tiras *t* y *s*, o bien son iguales, o bien definimos su relacion como la que existe entre los caracteres *t[i]* y *s[i]*, donde *i* es el primer caracter en el que *s* y *t* difieren (tengase tambien en cuenta la propiedad siguiente). Esto significa que

   "ABC" < "BCD" porque *'A' < 'B'*
   "ABC" > "ABB" porque *'C' > 'B'*
   "ABC" = "ABC" porque *no existe un tal* i

   lo cual corresponde a la idea de ordenacion utilizada habitualmente en los diccionarios.

2. Al comparar dos tiras de diferente longitud, se considerara que la mas corta esta *extendida* a la derecha con tantos espacios como sean necesarios para que las dos sean igual de largas. Por ejemplo,

   ```
   "Hola" = "Hola     "
   ```

   es cierto, pues, previamente a la comparacion, "Hola" se transforma en "Hola   ".

Entrada y salida de tiras

Los objetos de tipo *tira* pueden ser presentados y aceptados desde y hacia el exterior del programa, siguiendo esquemas similares a los explicados en la entrada y salida de enteros y caracteres.

- La instruccion *escribe* (o *escribe_linea*), aplicada a una (expresion de tipo) tira, se ha utilizado ya en el caso de tiras constantes: solo hay que añadir que se escriben tantos caracteres como tenga la tira en ese momento (y no tantos como su longitud maxima).
- La instruccion *lee* acepta tantos caracteres como la longitud maxima de la tira leida (o, si no los hay hasta fin de linea, los que haya), y forma con ellos una tira que asigna a la variable que se lee. Nunca se cambia de linea por efecto de una lectura de *tira*.
- Para efectuar un cambio de linea al leer datos, puede utilizarse la instruccion *lee_linea* (seguida o no de objetos a leer), cuyo efecto es ignorar (despues de leer los objetos designados) el resto de la linea y comenzar la siguiente.

Un ejemplo sencillo

Como ejemplo simple de lo explicado, presentamos un programa que, a partir de una linea de datos que contiene el nombre completo de una persona (que, para simplificar, supondremos compuesto de un nombre y dos apellidos, todos ellos sencillos), "entiende" estos datos y los presenta por separado.

Utilizaremos una variable

```
var c: tira[80];
```

que representara el nombre completo.

El unico paso no trivial del programa es la descomposicion de *c* en las tres palabras que lo forman. Para realizarla, deberemos identificar las posiciones inicial y final de cada una de las palabras de *c*. Esto lo haremos mediante un indice *i* que "recorrera" la tira buscando el principio y final de cada palabra.

```
programa Nombre_y_Apellidos es
  var c: tira(80);
  var i,n1,n2,a11,a12,a21,a22: entero;
  -- i "recorre" C; las demas variables enteras
  -- identifican los subindices que delimitan las palabras
haz
  lee c;
  i ← 0;
  si c[1] = ' ' entonces
    repite i ← i + 1; hastaque c[i] ≠ ' ';
  sino i ← 1;
  fin si;
  n1 ← i;
  repite i ← i + 1; hastaque c[i] = ' ';
  n2 ← i - 1;
  repite i ← i + 1; hastaque c[i] ≠ ' ';
  a11 ← i;
  repite i ← i + 1; hastaque c[i] = ' ';
  a12 ← i - 1;
  repite i ← i + 1; hastaque c[i] ≠ ' ';
  a21 ← i;
  repite i ← i + 1; hastaque c[i] = ' ';
  a22 ← i - 1;
  escribe_linea "Nombre: ",c[n1..n2];
  escribe_linea "Apellido1: ",c[a11..a12];
  escribe_linea "Apellido2: ",c[a21..a22];
fin programa;
```

Algoritmo 9. ***Descomposicion en Nombre y Apellidos:*** Notese que suponemos la existencia de al menos un espacio despues del segundo apellido.

Conviene resaltar que, a diferencia de otros programas, este realiza una sola operacion de lectura; el acceso posterior (y secuencial) a las componentes de lo leido no debe confundirse con ella.

Un ejemplo mas complejo: apariciones de una palabra en un texto.

El segundo programa, que puede tener mas aplicacion practica, intenta hallar el numero de veces que determinada palabra aparece en un texto. Los datos se presentan del siguiente modo: la palabra a buscar se encuentra al principio de una sola linea; las lineas siguientes contienen el texto, que se termina mediante un asterisco.

Utilizaremos el metodo de diseño descendente. Una primera aproximacion puede ser:

```
programa Cuenta_apariciones es
haz
  lee_la_palabra_a_buscar;
  repite
    lee_palabra_del_texto;
    si coinciden entonces
      incrementa_contador;
    fin si;
  hastaque se_termine_el_texto;
  escribe_resultados;
fin programa;
```

Por lo visto, necesitaremos como minimo las siguientes variables: dos tiras (que podemos llamar *p* y *q*) para representar la palabra que se busca y cada una de las demas, respectivamente; y un entero (que llamaremos *n*) para contar el numero de palabras que coinciden con la buscada.

```
var n: entero; p, q: tira(20);
```

Podemos refinar *lee_la_palabra_a_buscar* como

```
escribe_linea "Que palabra quieres buscar:";
lee p;
escribe_linea "Escribe ahora el texto, terminado por un asterisco";
lee_linea;
```

aprovechando para informar de que deseamos. Se notara la utilizacion de *lee* y *lee_linea*, forzada por el hecho de que esta ultima instruccion provoca, al ejecutarse, que el ordenador "pida" datos por pantalla, mientras que nosotros deseamos terminar la presentacion antes de que esto suceda.

Al intentar refinar *lee_palabra_del_texto*, la primera idea puede ser que las palabras seran las agrupaciones de caracteres delimitadas por blancos; esto se revela erroneo, sin embargo, al observar que una palabra puede estar inmediatamente seguida de un signo de puntuacion (Para simplificar, supondremos que los unicos signos de puntuacion son el punto, la coma, los dos puntos y el punto y coma). Asi, añadiremos una variable

```
c: caracter;
```

al conjunto de las declaradas, para examinar cada caracter, y escribiremos

```
q ← "";
repite
  q ← q || tira(c);
  lee c;
hastaque c = ' ' o c = '.' o c = ',' o c = ':' o c = ';' o c = '*';
si c = ' ' o c = '.' o c = ',' o c = ':' o c = ';' entonces
  repite lee c; hastaque c ≠ ' ';
fin si;
```

acumulando en *q* los caracteres que forman cada palabra, cuidando de no incluir en ella signo alguno de puntuacion, y suponiendo y manteniendo la hipotesis de que *c*, al terminar la accion, es el primer caracter de la siguiente palabra o bien el asterisco final (sera necesario modificar *lee_la_palabra_a_buscar*, añadiendo

```
lee c; n ← 0;
si c = ' ' entonces repite lee c; hastaque c ≠ ' '; fin;
```

a su refinamiento).

El resto del programa es sencillo:

coinciden se refinara evidentemente como $p = q$;
incrementa_contador como $n \leftarrow n + 1$;
escribe_resultados como *escribe_linea "La palabra '",p,"' aparece ",n," veces en el texto.";*
y *se_termine_el_texto* como $c = '*'$.

con lo que el programa final sera

```
programa Cuenta_apariciones es

  var n: entero; p, q: tira(20); c: caracter;

  accion lee_la_palabra_a_buscar haz
    escribe_linea "Que palabra quieres buscar?"; lee p;
    escribe_linea "Escribe ahora el texto, terminado por un asterisco";
    lee_linea; lee c; n ← 0;
    si c = ' ' entonces repite lee c; hastaque c ≠ ' '; fin;
  fin lee_la_palabra_a_buscar;

  accion lee_palabra_del_texto haz
    q ← "";
    repite
      q ← q || tira(c);
      lee c;
    hastaque c = ' ' o c = '.' o c = ',' o c = ':' o c = ';' o c = '*';
    si c = ' ' o c = '.' o c = ',' o c = ':' o c = ';' entonces
      repite lee c; hastaque c ≠ ' ';
    fin si;
  fin lee_palabra_del_texto;

  condicion coinciden haz vale p = q; fin;

  accion incrementa_contador haz n ← n + 1; fin;

  condicion se_termine_el_texto haz vale c = '*'; fin;

  accion escribe_resultados haz
    escribe_linea "La palabra '",p,"' aparece ",n," veces en el texto.";
  fin escribe_resultados;

haz
  lee_la_palabra_a_buscar;
  repite
    lee_palabra_del_texto;
    si coinciden entonces incrementa_contador; fin;
  hastaque se_termine_el_texto;
  escribe_resultados;
fin programa;
```

Algoritmo 10. Apariciones de una palabra en un texto

Ejercicios

1. Mostrar que, para cualesquiera tiras s y t, $long(s||t) = long(s) + long(t)$.
2. Considerar las transformaciones necesarias al Algoritmo 9 en la pagina 38 para
   a. evitar la suposicion de que un blanco sigue al segundo apellido
   b. evitar la repeticion de instrucciones **repite**.
3. Escribir un programa que detecte la existencia de palabras palindromicas [esto es, simetricas: como ALA ASA AMA ATA POP GAG u otras de mayor longitud].
4. Hacer lo mismo con frases palindromicas, como *dabale arroz a la zorra el abad.* Suponer un limite razonable a la longitud de las frases.
5. Dada una palabra y un texto, escribir todas las palabras que comiencen (o consistan) en esa.
6. Dadas dos palabras, escribir la posicion que la primera ocupa dentro de la segunda, o 0 si no lo hace. Ejemplos:

   rata matarratas → 6
   cocina fascina → 0

Los tipos *real* y *logico*. Expresiones

El tipo *real*

El lenguaje UBL proporciona dos tipos predefinidos para aplicaciones numericas: el tipo *entero* (que ya se ha estudiado) y el tipo *real*; intentan reflejar la estructura de los conjuntos Z y R de la Matematica, respectivamente. Los numeros reales pueden tener decimales (hasta un maximo dependiente de version) y estar expresados en notacion exponencial (lo cual permite abarcar un amplio rango de numeros aunque haya pocos digitos significativos). Se ha descrito ya la sintaxis de los literales reales; una declaracion de objeto real tiene la forma acostumbrada:

```
var r,s,t: real;
const pi = 3.1415926535;
```

Las operaciones aplicables a los numeros reales son la suma (+), resta (-), multiplicacion (*) y division (/). Se pueden mezclar reales y enteros en las operaciones aritmeticas; el resultado siempre es real. Asimismo, pueden dividirse dos enteros mediante el operador real "/"; en tal caso, el resultado es tambien real.

```
2.0 + 3.0 = 5.0
3.5 - 2 = 1.5
1.0/3.0 = 0.333333 (con mas o menos precision)
2/3 = 0.666666 (a diferencia de 2 div 3 = 0)
```

Las acciones predefinidas *lee*, *lee_linea*, *escribe* y *escribe_linea* pueden aplicarse a objetos y expresiones de tipo *real*. En lectura, se acepta cualquier formacion de caracteres que responda a la sintaxis de un entero o de un real; en escritura, se escribe el numero en notacion exponencial.

Precision de los objetos de tipo 'real'

Los numeros reales se representan en el ordenador con un numero fijo y acotado de digitos. Como consecuencia de ello, muchos de los resultados y teoremas matematicos dejan de ser validos:

- Supongamos que el numero de digitos que maneja el ordenador es 6. Entre los numeros 1.00000 y 1.00001 no hay ninguno intermedio *que tenga solo seis digitos*: los reales representables no forman un conjunto continuo.

- La igualdad $r+s=s$ no permite deducir $s=0$. Basta verlo con los numeros $r=1.0$ y $s=1.0E\text{-}10$:

  al sumarlos se obtiene

  ```
      r = 1.0000000000
  +   s = 0.0000000001
  ------------------------
  r+s = 1.0000000001   → truncado a 6 digitos: 1.00000 = 1 = r,
  ```

- $10/3*3 \neq 10$, ya que $10/3 = 3.33333...$, y al multiplicar por 3 se obtiene $9.99999... \neq 10$.

Como consecuencia, no se recomienda evaluar igualdades entre reales (por ejemplo, en condiciones), sino acotar su diferencia entre limites que se consideren oportunos: si en la instruccion

```
si x = y entonces escribe "La solucion es:",x; fin;
```

donde *x* e *y* son resultados reales de procesos distintos, se intenta evaluar si son iguales, es posible que nunca coincidan exactamente; es mejor substituir

$x = y$

por algo parecido a

$abs(x - y) < 1E\text{-}5$

Donde *abs* es una funcion predefinida que calcula el valor absoluto. Ademas, el exponente esta tambien limitado. Si se intenta leer un numero que exceda a su maximo, o alguna operacion produce un resultado que esta fuera del rango permitido, se produce un error.

Diferencias de calculo en la serie armonica.

La serie armonica se define como

$h_n = 1 + 1/2 + 1/3 + ... + 1/n$

El programa que sigue calcula h_n a partir de *n*, realizando la suma en orden ascendente primero (esto es, sumando 1/2 a 1, luego 1/3 al resultado y asi sucesivamente), y descendente despues. Es interesante ejecutarlo y observar las diferencias entre los resultados (teoricamente deberian ser identicos), que son relativamente mas importantes al crecer *n*.

```
programa serie_armonica es
  var n,i: entero;
  var suma_ascendente,suma_descendente: real;
haz
  escribe_linea "Vamos a calcular la suma de 1/i para i desde 1 hasta:";
  lee n;
  suma_ascendente ← 0; suma_descendente ← 0;
  i ← 1;
  repite
    suma_ascendente ← suma_ascendente + 1/i;
    i ← i + 1;
  hastaque i > n;
  i ← n;
  repite
    suma_descendente ← suma_descendente + 1/i;
    i ← i - 1;
  hastaque i < 1;
  escribe_linea "La suma en orden ascendente vale",suma_ascendente;
  escribe_linea "La suma en orden descendente vale",suma_descendente;
fin programa;
```

Algoritmo 11. Calculo de la Serie armonica

El tipo *logico*

El tipo *logico* se define por su conjunto de valores (solo tiene dos, que llamaremos *cierto* y *falso*) y las operaciones aplicables (**y**, **o**, **no**, ya estudiadas; **y entonces**, **o sino**, que se veran a continuacion). Se considera que

$falso < cierto$

y puede realizarse entrada y salida de valores y expresiones de tipo logico: se escribira `CIERTO` o `FALSO` segun sea el caso.

Suelen utilizarse variables de tipo *logico* en programas elaborados, para "recordar" el valor de determinada condicion en determinado momento de la ejecucion del programa. Debe evitarse utilizar este recurso si es posible.

$l \leftarrow x < 0$ y $(a > b$ o $a < c)$;
...
si l entonces ...

Mas adelante, al estudiar tipos de datos estructurados, se veran otras aplicaciones de este tipo.

Operadores logicos condicionales

Supongamos que deseamos verificar si las tres primeras letras de una tira t de caracteres son "ABC", y que expresamos esto mediante

si $T[1..3] = "ABC"$ ...

Se producira un error si, en el momento de evaluarse la condicion, la tira t no tiene una longitud mayor o igual que tres. Para prever este caso, podriamos intentar una solucion del estilo de

si $long(T) \geq 3$ **y** $T[1..3] = "ABC"$ ...

pero, dado que el significado del operador **y** especifica que deben evaluarse primero los dos operandos y despues hacer el **y** logico, nos encontramos con el mismo problema. Una solucion correcta seria

si $long(T) \geq 3$ **entonces si** $T[1..3] = "ABC"$ ...

aunque esto complicaria el programa, al añadir una nueva instruccion **si**.

Dado que este tipo de construccion es frecuente en programas complejos, definimos dos nuevas formas del calculo logico:

A **y entonces** B
A **o sino** B

que se evaluan como sigue:

A	B	A **y entonces** B
Falso	*No se evalua*	*Falso*
Cierto	*Falso*	*Falso*
Cierto	*Cierto*	*Cierto*

A	B	A **o sino** B
Cierto	*No se evalua*	*Cierto*
Falso	*Falso*	*Falso*
Falso	*Cierto*	*Cierto*

Figura 28. Operadores logicos condicionales

"No se evalua" significa que, de ser el valor de A el que indica la tabla, B (que puede ser una expresion compleja) no es calculado (ya que el valor de la expresion logica se conoce sin necesidad de hacerlo: *falso* y X es falso, independientemente de X, y *cierto* o X es siempre cierto)

Mediante estas formas, el fragmento anterior podra escribirse correctamente como

si $long(T) \geq 3$ **y entonces** $T[1..3] = "ABC"$ ...

limitando el calculo de $T[1..3] = "ABC"$ al caso en que $long(T) \geq 3$.

Expresiones

Podemos definir ahora con mas precision que entendemos por una *expresion*: en general, se compone de varios operandos y operadores; su *evaluacion* se realiza de acuerdo con las diferentes *prioridades* de esos operadores (es decir,

*a + b * c*

se interpreta como

*a + (b * c)* *[y no como (a + b) * c]*

debido a que el operador "*" tiene *mayor prioridad* -- es "mas fuerte" -- que "+"). Se supone tambien que todos los operadores son *asociativos por la izquierda*: esto es,

a + b + c + d

se interpreta como

((a + b) + c) + d

El formalismo sintactico definido en "Sintaxis y estilo" en la pagina 19 nos permitira describir la forma general de una expresion:

```
expresion = relacion { y relacion }
  | relacion { o relacion }
  | relacion { y entonces relacion }
  | relacion { o sino relacion }

relacion = expresion_simple [ operador_relacional expresion_simple ]

operador_relacional = = | ≠ | ≤ | ≥ | < | > | en

expresion_simple = [ + | - ] termino { operador_aditivo termino }

operador_aditivo = + | - | ||

termino = factor { operador_multiplicativo factor }

operador_multiplicativo = * | / | div | mod

factor = numero_sin_signo
  | tira
  | caracter_constante
  | conjunto
  | existencial
  | variable
  | seleccion_de_tira
  | invocacion_de_funcion
  | invocacion_de_condicion
  | conversion_de_tipo
  | "(" expresion ")"
  | no factor

seleccion_de_tira = variable "[" expresion [ ..expresion ] "]"
```

Figura 29. Sintaxis de una expresion

La prioridad de los distintos operadores queda reflejada en la sintaxis, asi como el papel de los parentesis para modificar el orden de evaluacion.

Los no terminales *conjunto*, *existencial* e *invocacion_de_funcion*, asi como el operador relacional "**en**", se estudiaran mas adelante; *variable* puede ser una variable (de las ya estudiadas) o asumir formas mas complejas (que tambien se estudiaran). *Invocacion_de_condicion*, puede ser, en su forma mas simple, el nombre de una condicion. En cuanto a *conversion_de_tipo*, se refiere a las

diversas posibilidades de conversion entre tipos que ofrece el lenguaje UBL. Se ha visto ya la conversion a tira:

tira(expresion_caracter)

Otros ejemplos se estudiaran.

Parametros y declaraciones locales

Entidades y declaraciones locales

En el proceso de diseño descendente, el refinamiento de un subprograma se realiza de un modo similar al refinamiento del programa principal: al hacerlo es probable que se necesiten nuevas declaraciones. Algunas de las entidades declaradas aparecen naturalmente como comunes a varios subprogramas, por ejemplo, en el caso de variables que contienen valores que deben ser manipulados en varias acciones; otras tienen un ambito logico puramente *local* (queremos decir: circunscrito a un solo subprograma). En este ultimo caso, declararlas en el programa principal causa problemas de estructura y reduce el espacio de identificadores disponible. El concepto de *declaracion local* sirve para evitar estos problemas, al permitir asociar cada declaracion con el ambito minimo que logicamente le corresponde:

> En el proceso de diseño de un programa nos encontramos en la necesidad de programar una accion *Intercambia* que intercambia los valores de las variables enteras *r* y *s*. Esto se realiza mediante la secuencia usual de instrucciones
>
> *t* ← *r*; *r* ← *s*; *s* ← *t*;
>
> que utiliza una variable auxiliar *t*. El uso de esta variable esta logicamente limitado a estas instrucciones, y puede resultar confuso declararla en algun lugar apartado; es mucho mas claro escribir
>
> **accion** *intercambia* **es**
> **var** *t: entero;*
> **haz**
> *t* ← *r*; *r* ← *s*; *s* ← *t*;
> **fin** *intercambia;*
>
> mediante una declaracion local.

La sintaxis de un subprograma con declaraciones locales es similar a la de un programa; la forma utilizada en capitulos anteriores es la abreviacion para el caso en que no hay entidades locales.

Accesibilidad y existencia de las entidades locales

Un objeto variable declarado localmente solo existe con cada ejecucion del subprograma que lo declara: en el ejemplo anterior, la variable *t* es "creada" cada vez que comienza la ejecucion de *intercambia* y "destruida" cada vez que termina esta. De este modo, es necesario asignar algun valor a las variables locales antes de utilizarlas: es un error creer que conservaran su valor entre una ejecucion y otra, ya que su existencia es discontinua. Ademas, es imposible referirse a una entidad declarada localmente desde un punto exterior al subprograma que la declara.

Reglas de reconocimiento de nombres

Los nombres de las entidades locales pueden coincidir con otros nombres declarados en el programa o en otros subprogramas. Para saber a que entidad se refiere un identificador se utiliza la siguiente convencion: se lo busca en la lista de declaraciones del subprograma en que aparece; si no se encuentra, se busca en la lista de declaraciones del subprograma (o, en su defecto, del programa) que incluye inmediatamente a aquel en el que se estaba buscando; y asi, sucesivamente, en todas las listas de todos los subprogramas que contienen sintacticamente al primero.

```
programa ejemplo es
  var i,j: entero; (* 1 *)
  accion a es
    var i: real; (* 2 *)
  haz
    (* cualquier referencia a I en el interior de la accion A accede a la declaracion {2},
       mientras que las referencias a J acceden a la declaracion {1} *)
  fin a;
haz -- programa
  (* cualquier referencia a I o j en el programa accede a la declaracion {1} *)
fin programa;
```

Como consecuencia, una declaracion local que utilice el mismo identificador que otra declaracion impide el acceso directo a la entidad representada por la segunda declaracion: se dice que la primera declaracion *esconde* a la segunda, y que la entidad declarada localmente *esconde* a la entidad homonima.

Recuerdese que, ademas, todas las entidades deben ser declaradas antes de utilizarse (esto es especialmente importante al decidir el orden de declaracion de los subprogramas). Si en algun caso se prefiere un orden distinto del exigido por esta regla, puede definirse la existencia de un subprograma sin mas que "anunciar" su presencia en el lugar en el que se requiere su declaracion, y escribir la declaracion completa mas abajo:

```
accion A; -- anuncia que A se declarara mas abajo, aunque ya puede utilizarse

accion B haz -- utiliza la accion A:
  ...
  A;
  ...
fin B;

accion A haz -- esta es la declaracion real de A
  ...
fin A;
```

Parametros

La potencia de las **acciones** y **condiciones** (y de los demas tipos de subprograma que se estudiaran) se amplia notablemente si consideramos la posibilidad de *parametrizar* sus efectos: la accion *Intercambia* del ejemplo anterior sera mucho mas util y general si se permite utilizarla en la forma

```
intercambia i,j;
```

o

```
intercambia k,p;
```

segun cuales sean las variables que se quieran intercambiar; una condicion *es_mayuscula* podra aplicarse en mas casos si admite un parametro:

```
es_mayuscula(c)
```

Un subprograma declara sus parametros inmediatamente despues del identificador que lo designa, y entre parentesis:

```
condicion es_numero(c: caracter) haz
  vale c ≥ '0' y c ≤ '9';
fin es_numero;
```

A nivel declarativo, los parametros se manejan mediante las mismas convenciones que las declaraciones locales.

Distinguiremos varios tipos de parametros:

lista_de_parametros = *parametros* {; *parametros*}

parametros = *parametros_por_copia*
| *parametros_por_nombre*
| *parametros_por_subprograma*

parametros_por_nombre = *parametros_por_variable* | *parametros_por_constante*

lista_de_identificadores = *identificador* {,*identificador*}

parametros_por_copia = *lista_de_identificadores: identificador;*

parametros_por_variable = **var** *lista_de_identificadores: identificador;*

parametros_por_constante = **const** *lista_de_identificadores: identificador;*

Figura 30. ***Sintaxis de la lista de parametros:*** Los *parametros_por_subprograma* se estudiaran mas adelante.

Por cada parametro declarado, toda invocacion de subprograma debe proporcionar un *argumento* que corresponde y se asocia con el correspondiente parametro. El modo en que se realiza esa asociacion depende del tipo de parametro.

- Un *parametro por copia* funciona como una variable local, que se inicializa con el valor del correspondiente argumento:

  Si declaramos

  ```
  accion raya(n: entero) haz -- escribe una raya formada con N guiones
    repite escribe '-'; n ← n - 1; hastaque n = 0;
  fin raya;
  ```

  una invocacion como

  ```
  raya 30;
  ```

  ejecuta la accion *raya* despues de asignar el valor del argumento (*30* en este caso) al parametro *n*.

  El argumento puede ser *cualquier expresion* del mismo tipo que el parametro (o de tipo *entero* si el parametro es *real*).

- Un *parametro por variable* funciona como un *sinonimo* para la variable proporcionada como argumento:

  La accion *intercambia* se generaliza mediante parametros por variable:

  ```
  accion intercambia(var r,s: entero) es
    var t: entero;
  haz
    t ← r; r ← s; s ← t;
  fin intercambia;
  ```

  de modo que al utilizarla

  ```
  intercambia i,j;
  ```

  r y *s* funcionan como sinonimos locales de los argumentos proporcionados: la ejecucion de la ultima instruccion sera equivalente a

  ```
  t ← i; i ← j; j ← t;
  ```

  que tiene el efecto buscado.

  El argumento debe ser *una variable del mismo tipo* que el parametro, que lo denota durante la ejecucion del subprograma.

- Un *parametro por constante* es un *sinonimo* que funciona como un objeto *constante* con el valor de la expresion suministrada como argumento. Consecuentemente, no es posible alterar dentro

de un subprograma (mediante una asignacion u otros medios) el valor de un parametro constante, y es un error establecerlo como sinonimo de un argumento que va a ser variado durante la ejecucion del subprograma.

```
var x: entero:

accion A haz x ← x + 1; fin; -- modifica x

accion B(const C: entero) haz
  A; -- erroneo si se invoco B x;
fin B;
```

Suelen utilizarse parametros por constante cuando su tipo es estructurado (p. ej., tiras, o **tablas** o **tuplas**, que se estudiaran) y ocupa mucho espacio (ya que entonces el compilador no realiza copia alguna del valor, con lo que se ahorra memoria) o para informar al compilador de que se desea detectar cualquier intento de alterar su valor. Los correspondientes argumentos pueden ser expresiones o variables, indistintamente.

Ejemplos de utilizacion de parametros pueden encontrarse en los siguientes capitulos.

Efecto de Alias

Diremos que se produce un *efecto de alias* cuando, debido a las asociaciones introducidas por los parametros por nombre o por otras causas, el mismo objeto es conocido en un punto del programa mediante dos nombres distintos:

```
var x: entero;
accion A(var z: entero) haz
  -- en este punto, y tras la invocacion "A x;",
  -- Z y X son nombres distintos para el mismo objeto.
fin A;

haz
  A x;
```

Dado que normalmente se diseñan los programas haciendo la suposicion implicita de que dos identificadores distintos representan objetos distintos, el efecto de alias puede producir resultados inesperados, y debe evitarse o preverse si es posible.

Un ejemplo que ademas ilustra un principio mas general: otro modo de escribir la accion *intercambia*, en este caso sin variables locales, es

```
accion intercambia(var r,s: entero) haz
  -- r = r0, s = s0
  r ← s - r;     -- r = s0 - r0
  s ← s - r;     -- s = s0 - (s0 - r0) = r0
  r ← r + s;     -- r = (s0 - r0) + r0 = s0
  -- r = s0, s = r0
fin intercambia;
```

Los comentarios constituyen una demostracion de la validez del metodo. Se observara que el calculo substituye a los datos, es decir, las operaciones de suma y resta substituyen a la variable local; esto es general: en muchos casos, *algoritmos y datos son intercambiables*.

Si ahora invocamos

```
intercambia x,x;
```

se produce un efecto de alias, con el resultado imprevisto de anular el valor de x (observese que esto no sucede con la otra version de *intercambia*).

Sintaxis completa de las declaraciones de subprograma y sus invocaciones

declaracion_de_subprograma = *cabecera bloque* | *cabecera;*

bloque =
  [**es**
    {*declaracion*}]
  haz
    {*instruccion*}
  fin [*identificador*];

cabecera =
    accion *identificador* ["("*lista_de_parametros*")"]
  | **condicion** *identificador* ["("*lista_de_parametros*")"]
  | **funcion** *identificador* ["("*lista_de_parametros*")"]: *identificador*
  | **secuencia** *identificador* ["("*lista_de_parametros*")"]: *identificador*

Figura 31. ***Sintaxis de las declaraciones de subprograma:*** Las funciones y secuencias se estudian en los capitulos siguientes; la forma *cabecera;* se refiere al "anuncio" de declaracion mencionado mas arriba.

argumentos = *argumento* {,*argumento*}

argumento = *expresion*

Figura 32. ***Sintaxis de la lista de argumentos***

instruccion_de_invocacion = *identificador* [*argumentos*];

Figura 33. ***Sintaxis de la instruccion de invocacion a una*** **accion**

invocacion_de_condicion = *identificador* ["("*argumentos*")"]

Figura 34. ***Sintaxis de una invocacion de*** **condicion**

Funciones

Introduciremos un nuevo tipo de subprograma que nos permitira escribir abstracciones de evaluacion.

Necesidad de las Funciones.

En el proceso de diseño descendente es necesario en ocasiones disponer de la capacidad de suponer la existencia de determinadas operaciones de transformacion de datos (p. ej., la raiz cuadrada de un numero, la inversa de una matriz, la verdad de si un entero es impar). A estas transformaciones las llamaremos *funciones*; un ejemplo de ellas es la entidad predefinida *impar*, que aplica valores enteros en valores logicos. La mayoria de funciones tienen parametros por copia o constante que corresponden a las variables independientes de las funciones matematicas y a partir de los cuales se evalua el resultado. Estos parametros se declaran con la funcion:

lo que matematicamente se expresa

$$\begin{array}{l} S \xrightarrow{\quad f \quad} T \\ x \longrightarrow f(x) \end{array}$$

se escribira en UBL como

funcion *f (x: S): T*

por ejemplo,

funcion *impar(n: entero): logico*

y se utilizan como variables dentro de la propia funcion. Un ejemplo puede ayudar a comprender esto:

```
funcion media(a,b: real): real haz
  vale (a + b) / 2;
fin media;
```

es una *declaracion de funcion*, que introduce una abstraccion de subprograma para calcular la media de cualesquiera valores reales *a* y *b*. Una vez definida, puede utilizarse en cualquier punto del programa mediante una *invocacion de funcion*:

```
m ← media(z1,z2);
...
escribe "La media es: ",media(x,3.0);
```

media(z1,z2) o *media(x,3.0)* son invocaciones de funcion. *A* y *b* no son mas que *parametros* o "variables falsas" que toman los valores indicados entre parentesis en cada invocacion: como si, antes de ejecutar las instrucciones asociadas a *media*, se hiciese

```
a ← z1; b ← z2;
```

en el primer caso, y

```
a ← x; b ← 3.0;
```

en el segundo.

invocacion_de_funcion = identificador ["("argumentos")"]

***Figura 35. Sintaxis de una invocacion de* funcion**

Al igual que en las **condiciones**, la ejecucion de una funcion debe terminar con la de una instruccion **vale**, que en este caso debera designar una expresion del tipo declarado con la funcion e indicara el valor que toma su invocacion.

instruccion_vale = **vale** *expresion;*

***Figura 36. Sintaxis de la instruccion* vale**

Pueden declararse tantos parametros como sea necesario; los tipos de estos pueden o no coincidir.

Algunas funciones predefinidas; ejemplos simples

Algunas de las funciones predefinidas que pueden utilizarse con los tipos estudiados son:

- **funcion** *abs(x: T): T;*, donde *T* es *real* o *logico*, calcula el valor absoluto de una expresion.
- **funcion** *impar(n: entero): logico;* **vale** *n* **mod** *2* ≠ *0.*
- **funcion** *pred(n: entero): entero;* es el *PREDecesor* de *n*; esto es, **vale** *n - 1.*
- **funcion** *suc(n: entero): entero;* es el *SUCesor*: **vale** *n + 1.*
- **funcion** *trunc(r: real): entero;* trunca un numero *real* (o sea, ignora sus decimales) para considerarlo *entero.*

Los siguientes son ejemplos de funciones; todos ellos son evidentes o auto-documentados:

```
funcion min(a,b: real): real haz -- calcula el minimo de A y B
  si a < b entonces vale a;
  sino vale b;
  fin si;
fin min;

funcion media(a,b,c: real): real haz -- calcula la media de A, B y C
  vale (a + b + c) / 3;
fin media;

funcion cubo(r: entero): entero haz vale r*r*r; fin;
```

La declaracion de tipo para tiras

La sintaxis de una declaracion de funcion, asi como la de la lista de parametros, especifica que los tipos que aparecen deben escribirse en forma de identificadores. Para poder escribir funciones que operen sobre tiras (y sobre tipos mas elaborados que se estudiaran mas adelante) es necesario utilizar lo que llamaremos una *declaracion de tipo*:

tipo *identificador* **es** *tira(expresion_constante);*

***Figura 37. Sintaxis de una declaracion de tipo* tira**

Una declaracion de este estilo amplia el repertorio de tipos disponibles, añadiendo *identificador* al de los ya existentes:

```
tipo palabra es tira(10);
```

permite utilizar *palabra* como se utiliza *entero* o *real*. Las declaraciones

```
var p1,p2: palabra;
```

y

```
var p1,p2: tira(10);
```

son entonces equivalentes, con la ventaja en el primer caso de que el identificador *palabra* puede ser mnemotecnico.

Funcion &indice.

Una operacion frecuente al manejar tiras es la de averiguar si una tira s "forma parte" de otra tira t, en el sentido de que existen indices i y j tales que $t[i..j]=s$. Normalmente, interesa saber el valor del indice i (ya que j se obtiene como $i+long(s)-1$). Supuesta la declaracion del tipo *palabra* escrita anteriormente, buscamos una

```
funcion indice(s,t: palabra): entero
```

que evalue i; para prever el caso en que s no esta contenida en t, y por razones de simetria, convendremos en que

$$indice(s,t) = 0 \iff S \text{ no forma parte de } T$$

Una primera aproximacion a la escritura de la funcion podria ser

```
funcion indice(s,t: palabra): entero es
  var i,j: entero;
haz
  i ← 1; j ← long(s);
  si j > long(t) entonces vale 0;
  sino
    repite
      si t[i..j] = s entonces vale i;
      sino i ← i + 1; j ← j + 1;
      fin si;
    hastaque j > long(t);
    vale 0;
  fin si;
fin indice;
```

Algoritmo 12. *Indice para tiras de caracteres*

siguiendo la explicacion del significado de la funcion. Cada iteracion compara la tira s con una subtira de t, lo cual suele ser una operacion costosa para el interprete. Otra version, basada en buscar primero un caracter de t que coincida con $s[1]$ y mirar luego si el resto tambien coincide, podria ser mas economica, y se plantea como ejercicio.

Calculo de la Raiz Cuadrada de un numero Real mediante el metodo de Biparticion

Se trata de escribir una

```
funcion raiz(r: real): real
```

que calcule la raiz cuadrada de un numero real no negativo r, esto es, que halle x tal que

$$x = raiz(r)$$

o, lo que es lo mismo,

$$x^2 = r$$

Para ello utilizaremos el metodo iterativo llamado de *biparticion*: supuestos halladas dos aproximaciones al valor de x, x_0 y x_1, tales que x_0 "no llega" a ser la raiz cuadrada y x_1 "se pasa",

$$x_0^2 < r$$
$$x_1^2 > r$$

consideraremos el valor $x_2=(x_0+x_1)/2$, la media aritmetica de x_0 y x_1, como una nueva aproximacion. Puede suceder que se verifique exactamente

$$x_2^2 = r$$

en cuyo caso $x=x_2$ es el valor buscado, o bien que x_2 "no llegue" o "se pase" de la x buscada. En ese caso, substituiremos x_0 (o x_1) por x_2, con lo que habremos obtenido un par de valores en las mismas condiciones que al principio, pero que en este caso *diferiran tan solo en la mitad* de lo que diferian antes. Aplicando el proceso tantas veces como sea necesario se obtendra, o bien la solucion exacta en algun momento del proceso, o bien una sucesion de intervalos encajados convergentes hacia un punto, que, de nuevo, sera la solucion exacta. Teniendo en cuenta la finitud de la precision de los numeros reales, es de esperar que la diferencia entre x_0 y x_1 llegue a ser menor el incremento minimo que permitiria distinguirlos en el ordenador; lo que permitira detener el proceso, como maximo, cuando $x_0 = x_1$ o $x_0 = x_2$.

Como ejemplo calcularemos la raiz cuadrada de 2 mediante este metodo: $x_0=1$ y $x_1 = 2$ pueden ser buenas aproximaciones iniciales. Esto nos da $x_2 = 1.5$, que es superior a la raiz cuadrada buscada, por lo que reemplaza a x_1, obteniendo $x_0=1$ y $x_1=1.5$ como nuevo par de valores. Mostramos aqui los primeros pasos del calculo:

x_0	x_1	x_2	x_2^2
1.0	*2.0*	*1.5*	*2.25*
1.0	*1.5*	*1.25*	*1.5625*
1.25	*1.5*	*1.375*	*1.890625*
1.375	*1.5*	*1.4375*	*2.06640625*
1.375	*1.4375*	*1.40625*	*1.97753...*

Figura 38. Fragmento del calculo de la raiz de 2 por Biparticion

se observara como x_0, x_1 y x_2 convergen hacia el valor buscado.

Para determinar el par de valores inicial, bastara observar que la raiz de r es menor que r si r es mayor que 1, y viceversa: de modo que 1 y r (o r y 1) seran siempre una aproximacion inicial correcta (que sea buena no importa mucho, ya que el proceso de biparticion se encarga de mejorarla).

```
funcion raiz(r: real): real es
  var x0, x1, x2: real;
haz
  decide
    cuando r = 0 => vale 0;
    cuando r = 1 => vale 1;
    cuando otros =>
      x0 <- 1; x1 <- 1;
      si r > 1 entonces x1 <- r; sino x0 <- r; fin;
      x2 <- (x1 + x0) / 2;
      repite
        decide
          cuando x2 * x2 > r => x1 <- x2;
          cuando x2 * x2 < r => x0 <- x2;
          cuando otros       => vale x2;
        fin decide;
        x2 <- (x0 + x1) / 2;
      hastaque (x2 = x0) o (x2 = x1);
      vale x2;
  fin decide;
fin raiz;
```

Algoritmo 13. Raiz cuadrada por Biparticion

Ejercicios

1. Reescribir la funcion *indice* tal como se indica e n el texto.

2. Escribir una

   funcion *seno(x: real): real*

   Utilizando algun desarrollo en serie del seno. Si no se conoce ninguno, reducir x al primer hemiciclo y aplicar el de Taylor:

$$sen\ x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

Secuencias

El tratamiento secuencial de datos se presenta tan frecuentemente en programacion, que hemos creido conveniente introducir un tipo de subprograma para manejarlo. Paralelamente a las **funciones**, que producen un solo valor, las **secuencias** producen una serie o coleccion secuencial de ellos. Estas colecciones pueden ser tratadas de dos modos: aplicando algun tratamiento a cada uno de sus valores o buscando cual de sus elementos cumple determinada condicion; ademas, es posible restringir el tratamiento solo a subcolecciones de la secuencia.

Un ejemplo

Sea *t* una tira de caracteres, que deseamos imprimir "en vertical", esto es, un caracter por linea. En una primera aproximacion, podemos formular esto asi

```
para c en caracteres_de_T haz
  escribe_linea c;
fin para;
```

utilizando la instruccion **para**; debe leerse

> "Para todo *c* perteneciente a (o: tomando *c* los valores de) la secuencia de los caracteres que integran *t*, escribe una linea conteniendo *c*."

Caracteres_de_T es una *secuencia* de valores, que debe ser refinada segun el metodo usual de diseño descendente:

```
secuencia caracteres_de_T: caracter es
  var i: entero;
haz
  si long(T) > 0 entonces
    i ← 1;
    repite
      produce T[i];
      i ← i + 1;
    hastaque i > long(T);
  fin si;
fin caracteres_de_T;
```

El esquema es obvio; la instruccion **produce** indica cuales son los valores que forman la secuencia.

Con mayor detalle: la ejecucion de la instruccion **para** *activa* la secuencia *caracteres_de_T*, empezando a ejecutar sus instrucciones. Cada vez que se encuentra una instruccion **produce**, el valor calculado en la instruccion se asigna a la variable *c* (que suponemos, con la tira *t*, declarada en algun lugar), se ejecutan las instrucciones subordinadas a la **para**, y se continua la ejecucion de la secuencia con la instruccion siguiente a la **produce**.

Diremos que una secuencia esta *activa* desde que empieza su ejecucion hasta que *termina* (lo cual puede suceder al llegar al final de la secuencia, mediante la ejecucion de una instruccion **acaba** o implicitamente al evaluar un existencial). Al ejecutar una instruccion **produce**, pasa a ejecutarse la parte de programa que activo la secuencia, pero esta permanece activa: diremos que esta *suspendida*, y que su ejecucion *continua* al seguir ejecutandose sus instrucciones. Mientras una secuencia esta activa, las variables locales (y parametros) conservan su valor, y solo lo pierden al terminarse (pero no al suspenderse).

Las secuencias pueden tener parametros y declaraciones locales (como todos los tipos de subprograma).

Las secuencias predefinidas *asc* y *desc*

- *asc(i,j)*, donde *i* y *j* son expresiones enteras, denota la secuencia de valores enteros *asc*endientes entre *i* y *j* (secuencia que puede ser vacia, en el caso de que $j < i$); similarmente,
- *desc(i,j)*, donde *i* y *j* son expresiones enteras, denota la secuencia de valores enteros *desc*endientes entre *i* y *j* (secuencia que puede ser nula, en el caso de que $j > i$).

Mediante el uso de la secuencia predefinida *asc*, el ejemplo anterior podria escribirse

```
para i en asc(1,long(T)) haz
  escribe_linea T[i];
fin para;
```

sin programar ninguna secuencia y ganando en claridad.

Al estudiar otros tipos de datos se veran otras aplicaciones de las secuencias predefinidas *asc* y *desc*.

La instruccion 'para'

La forma general de la instruccion **para** es

```
instruccion_para =
  para variable en iteracion [talque condicion] haz
    instruccion
    {instruccion}
  fin [para];

  iteracion = identificador ["("argumentos")"]
```

Estilo: ademas del sugerido en la sintaxis, puede utilizarse:

para *variable* **en** *iteracion* [**talque** *condicion*] **haz** *instruccion(es);* **fin**;

***Figura 39. Sintaxis y estilo de la instruccion* para**

Su significado ya se ha explicado en parrafos anteriores. La parte **talque**, opcional, sirve para limitar la ejecucion de las instrucciones subordinadas a la verificacion de determinada condicion (que suele ser alguna propiedad de los elementos de la secuencia): en el ejemplo anterior, si deseamos suprimir los blancos al imprimir los caracteres de *t*, escribiremos:

```
para i en asc(1,long(T)) talque T[i] ≠ ' ' haz
  escribe_linea T[i];
fin para;
```

lo cual evidentemente equivale a

```
para i en asc(1,long(T)) haz
  si T[i] ≠ ' ' entonces
    escribe_linea T[i];
  fin si;
fin para;
```

pero es mas compacto y legible.

"**para**" puede leerse "para todo" o "para cada"; "**en**", "perteneciente a". La forma general de la instruccion esta tomada del cuantificador universal utilizado en Logica.

Existenciales

Variacion sobre el mismo ejemplo: lo que deseamos ahora es. no escribir la tira, sino el valor del primer indice en *t* (si existe) tal que *T[i]* sea un espacio en blanco. Para ello, podemos utilizar la construccion

```
si existe i en asc(1,long(T)) talque T[i] = ' ' entonces
  escribe_linea i;
fin si;
```

Existe es una forma de condicion general que se aplica sobre secuencias y tiene el efecto adicional de, en el caso de que valga *cierto*, asignar a la variable mediante la que se pregunta la existencia el primer valor de la secuencia que la satisfaga.

Otro ejemplo: una forma (ineficiente) de calcular el primer entero cuyo cuadrado supera al numero 1000 es

```
si existe i en asc(1,100) talque i*i > 1000 entonces
  escribe_linea i;
fin si;
```

En este caso, se utiliza la instruccion **si** sabiendo de antemano que la condicion es cierta, y con el unico objetivo de lograr que la evaluacion del existencial asigne a *i* el valor correcto.

Presentamos tambien la siguiente condicion que evalua (de un modo muy poco eficiente) si un entero es primo o no:

```
condicion primo(p: entero) es
  var q: entero;
haz
  vale no existe q en asc(2,p-1) talque p mod q = 0;
fin primo;
```

Algoritmo 14. Para saber si un entero es primo

Los existenciales estan tomados tambien a imagen de los cuantificadores logicos; su sintaxis es

existencial = **existe** *variable* **en** *iteracion* [**talque** *condicion*]

Figura 40. Sintaxis de un existencial

La *variable* que aparece en la sintaxis de **para** y **existe** se conoce como *variable de control.*

Debe tenerse en cuenta que la utilizacion del predicado **existe** puede implicar gran cantidad de calculos para evaluar un solo dato, por lo que es recomendable medir cuidadosamente su efecto antes de utilizarlo.

Otro ejemplo. Sintaxis de la instruccion produce y de las declaraciones de secuencia

El criterio de variacion ofrecido por la secuencia *asc* es bastante limitado: puede pensarse un uno mas amplio, que considere incrementos cualesquiera, y no necesariamente unitarios:

ascen(i,j,k)

representa el conjunto de valores i, $i+k$, $i+2k$, ..., hasta el ultimo $i+nk$ que no supera a j. Una manera de programar tal secuencia es

```
secuencia ascen(i,j,k: entero): entero haz
  si i ≤ j entonces
    repite
      produce i;
      i ← i + k;
    hastaque i > j;
  fin si;
fin ascen;
```

La sintaxis de la instruccion **produce** es

instruccion_produce = **produce** *expresion;*

Figura 41. Sintaxis de la instruccion **produce**

Un programa complejo: el justificador de textos

Como muestra del poder de abstraccion que proporcionan las secuencias, desarrollaremos un programa bastante complejo: se trata de "editar" un texto, encolumnandolo de modo que quede bien alineado (por la izquierda y por la derecha), a base de distribuir espacios en blanco entre las palabras que forman una linea. Se tomara como dato el texto mismo, escrito en cualquier forma, y se producira como resultado el texto bien impreso.

Utilizaremos la condicion predefinida *fdf* (abreviacion de Fin De Fila) para determinar si hemos llegado o no al final de los datos.

Analisis del problema:: los datos se componen de caracteres, que se presentan secuencialmente; las agrupaciones de caracteres delimitadas por blancos formas palabras, que tambien se presentan secuencialmente; por ultimo, varias palabras se combinan con espacios en blanco para construir la secuencia final de lineas, que es lo que debemos imprimir.

Formulacion del algoritmo:: abandonando por una vez el esquema de diseño descendente, y ciñendonos al analisis anterior, escribiremos una secuencia

```
secuencia caracteres: caracter es
  var c: caracter;
haz
  repite
    lee c;
    produce c;
  hastaque fdf;
fin caracteres;
```

que producira los caracteres del texto; a continuacion, y tras declarar

```
const max_palabra = 20;
tipo palabra es tira(max_palabra);
```

para poder declarar palabras (que se suponen de longitud maxima 20), escribiremos la secuencia

```
secuencia palabras: palabra es
  var c: caracter; p: palabra;
haz
  p ← "";
  para c en caracteres haz
    si c = ' ' entonces
      si p ≠ "" entonces
        produce p;
        p ← "";
      fin si;
    sino
      p ← p || tira(c);
    fin si;
  fin para;
fin palabras;
```

de las palabras del texto. La variable *p* actua como un acumulador, "recibiendo" los caracteres individuales hasta formar una palabra, momento en el que se **produce** *p* como elemento de la serie de palabras. Se notara que la señal para terminar una palabra es la aparicion de un blanco, por lo que sera conveniente añadir la instruccion

```
produce ' ';
```

al final de la secuencia *caracteres* (lo cual significa suponer un blanco mas al final del texto: no altera el resultado) para garantizar el funcionamiento correcto de *palabras*.

Finalmente, y utilizando un esquema similar al de *palabras*, declararemos

```
const max_linea = 80;
tipo linea es tira(max_linea);
```

y escribiremos la secuencia de las lineas:

```
secuencia lineas: linea es
  var p: palabra; l: linea;
haz l ← "";
  para p en palabras haz
    decide
      cuando long(l) + long(p) + 1 > max_linea ⇒
        produce margina(l);
        l ← p;
      cuando l = "" ⇒ l ← p;
      cuando otros ⇒ l ← l || " " || p;
    fin decide;
  fin para;
  produce l;
fin lineas;
```

Margina es una funcion que aplica lineas en lineas y se limita a distribuir blancos hasta lograr la alineacion correcta de las palabras que las forman. En este caso, *l* es el acumulador, y se separan las palabras (hasta su marginacion) mediante un blanco. La ultima linea se produce tal cual, sin utilizar la funcion de marginacion. La condicion en la instruccion **si** calcula si la palabra "cabe" o no dentro de la linea: si no cabe, se produce la linea (bien formada) y se comienza una nueva linea que consta al principio de una sola palabra; si cabe, basta añadirla.

Suponiendo resuelta la funcion *Margina*, el programa se reducira a

```
para l en lineas haz
  escribe_linea l;
fin para;
```

lo cual es trivial.

Para conseguir, a partir de una linea, obtener otra de exactamente 80 caracteres con los blancos bien distribuidos, utilizaremos el siguiente metodo: calcularemos el numero de caracteres que le faltan a la linea para llegar a 80:

```
sobrante ← max_linea - long(l);
```

En el caso de que sea 0. la linea ya esta bien ajustada: en otro caso. calculamos el numero de espacios en blanco que tiene la linea sin ajustar (esto es, el numero de lugares en los cuales insertar blancos):

```
lugares ← 0;
para c en caracteres talque c=' ' haz
  lugares ← lugares + 1;
fin para;
```

donde *caracteres* es una secuencia local a la funcion que proporciona los caracteres de la linea. Finalmente, distribuiremos los caracteres sobrantes entre los lugares designados, cuidando de hacerlo exactamente:

```
blancos ← sobrante div lugares;
sobrante ← sobrante mod lugares;
resultado ← "";

para c en caracteres haz
  si c ≠ ' ' entonces
    resultado ← resultado || tira(c);
  sino
    si sobrante > 0 entonces
      para j en asc (0,blancos + 1) haz
        resultado ← resultado || " ";
      fin para;
      sobrante ← sobrante - 1;
    sino
      para j en asc(0,blancos) haz
        resultado ← resultado||" ";
      fin para;
    fin si;
  fin si;
fin para;
```

La linea resultante se formara en la variable *resultado*.

```
programa Justificador es

  const max_palabra = 20; const max_linea = 80;
  tipo palabra es tira(max_palabra); tipo linea es tira(max_linea);
  var l: linea;

  secuencia caracteres: caracter es
    var c: caracter;
  haz
    repite lee c; produce c; hastaque fdf;
  fin caracteres;

  secuencia palabras: palabra es
    var c: caracter; p: palabra;
  haz
    p ← "";
    para c en caracteres haz
      si c = ' ' entonces si p ≠ "" entonces produce p; p ← ""; fin;
      sino p ← p || tira(c);
      fin si;
    fin para;
  fin palabras;

  funcion margina(l: linea): linea es
    secuencia caracteres: caracter es var j: entero;
    haz
```

```
    para j en asc(1,long(l)) haz produce l[j]; fin;
  fin caracteres;
  var resultado: linea; c: caracter; lugares, sobrante: entero; j,blancos: entero;
haz
  sobrante ← max_linea - long(l);
  si sobrante = 0 entonces vale l; fin;
  lugares ← 0; para c en caracteres talque c=' ' haz lugares ← lugares + 1; fin;
  blancos ← sobrante div lugares; sobrante ← sobrante mod lugares; resultado ← "";

  para c en caracteres haz
    si c ≠ ' ' entonces resultado ← resultado || tira(c);
    sino
      si sobrante > 0 entonces
        para j en asc (0,blancos + 1) haz resultado ← resultado || " "; fin;
        sobrante ← sobrante - 1;
      sino para j en asc(0,blancos) haz resultado ← resultado||" "; fin;
      fin si;
    fin si;
  fin para;
  vale resultado;
fin margina;

secuencia lineas: linea es
  var p: palabra; l: linea;
haz   l ← "";
  para p en palabras haz
    decide
      cuando long(l) + long(p) + 1 > max_linea ⇒ produce margina(l); l ← p;
      cuando l = "" ⇒ l ← p;
      cuando otros ⇒ l ← l || " " || p;
    fin decide;
  fin para;
  produce l;
fin lineas;

haz
  para l en lineas haz escribe_linea l; fin;
fin programa;
```

Algoritmo 15. Justificador de textos

Ejercicio

Extender el Algoritmo 15 del siguiente modo: los datos contendran, ademas del texto a editar, lineas consistentes en instrucciones sobre la apariencia que debe tomar el texto: por ejemplo, en

```
final de linea de texto
.es
principio de siguiente linea de texto
```

la linea ".es" indicara al programa que debe dejarse una linea en blanco entre la ultima linea y la siguiente.

Las instrucciones estaran siempre en linea aparte, comenzaran con un punto y constaran de dos letras siguiendo al punto y posiblemente alguna otra indicacion. El programa debera "entender" como minimo las siguientes:

.es n (ESpacio) deja N lineas en blanco.

.pa (PAgina) pasa de pagina

.si (+n|-n) (Sangrado Izquierdo) corre el margen izquierdo N espacios a la derecha (+N) o a la izquierda (-N).

.sd (+n|-n) (Sangrado Derecho) funciona como .SI para el margen derecho.

.ne (empieza NEgrita) empieza a escribir en negrita.

.su (empieza SUbrayado) empieza a escribir subrayado

.no (empieza NOrmal) empieza a escribir normal

Todas las instrucciones provocan cambio de linea. Los efectos de subrayado y negrita pueden conseguirse mediante el uso de *caracteres de control*: el primer caracter de cada linea se utilizara para determinar como se imprime la linea y no como parte de la linea. Asi, una linea que contenga

```
1Primera pagina
```

provocara la impresion de "Primera pagina" *en la primera linea de la siguiente pagina.* Los caracteres de control utilizables son:

' ' el espacio en blanco fuerza la impresion de la linea a continuacion de la anterior (caso normal).

'1' fuerza la impresion de la linea a principio de la siguente pagina (salto de pagina)

'+' fuerza la impresion de la linea *sobre* la anterior (sobreimpresion)

El formato de las paginas sera de 80 columnas impresas (sin contar la de control) y 60 lineas; las paginas deben estar numeradas.

Como indicacion para resolver el problema, puede considerarse la **secuencia** de instrucciones del texto.

Tipos Subrango y Enumerados

Tipos enumerados

Al estudiar conjuntos en Matematicas elementales, suelen explicarse dos formas de describirlos:

- por *extension*, esto es, citando cada uno de los elementos que forman el conjunto

  $C = \{1,3,5,7,9\}$

- por *comprension*, esto es, dando una propiedad que caracteriza a sus elementos

  $C = \{ x \text{ en } N \mid x \text{ es impar y } x < 10 \}$

Introduciremos formas de definir nuevos tipos de datos analogas a las definiciones de conjuntos por extension.

Una declaracion de la forma

tipo *T* **es** $\{i_1,i_2, \ldots, i_n\}$;

donde $i_1, i_2, \ldots i_n$ son identificadores, introduce *T* como nuevo tipo. Diremos que *T* es un *tipo enumerado no ordenado*. Un objeto de ese tipo

var *x: T;*

puede tomar cualquiera de los valores $i_1, i_2, \ldots, i_n$, y solamente esos: son *constantes de tipo T*. Puede hacerse

$x \leftarrow i_1$;
$x \leftarrow i_n$;

Obviamente, las dos declaraciones anteriores equivalen a la

var $x: \{i_1,i_2, \ldots, i_n\}$;

Al ser *T* un nuevo tipo, no es posible asignar expresiones de tipo *T* a objetos de otro tipo, ni viceversa.

Las operaciones aplicables a los objetos de un tipo enumerado no ordenado son:

- La asignacion y la comparacion por (no) igualdad.
- **funcion** *ord(x: T): entero;* es lo que se llama el *ordinal* de una expresion de tipo *T*. Aplicado a i_k vale *k-1*; es decir da el lugar que ocupa en la lista, empezando a contar desde 0.
- La *conversion de tipo T(k)*, donde *k* es una expresion entera, es la funcion inversa de la *ordORD*: $T(k-1)=i_k$, $T(ord(i_k))=i_k$, $ord(T(k)) = k$. Es un error calcular *T* de $k < 0$ o $k \geq n$.
- Las secuencias predefinidas *asc* y *desc* funcionan para expresiones de estos tipos: $asc(i_k,i_l)$ es $i_k, i_{k+1}, \ldots i_{l-1}, i_l$ (siempre que $k \leq l$), y al reves con *desc*.
- Puede realizarse entrada y salida con objetos y expresiones de tipo enumerado: se escribira el identificador que denota el valor deseado.

Si en la declaracion del tipo substituimos las llaves por parentesis

tipo *T* **es** *(i_1,i_2, ..., i_n)*;

obtenemos lo que llamaremos un *tipo enumerado ordenado*, que añade a las operaciones del tipo no ordenado todas las comparaciones (con el criterio $i_j < i_k \Leftrightarrow j < k$) y las

- **funcion** *suc(x: T): T;* es el SUCesor de *x* en la lista declarada: $suc(i_1) = i_2$, etc.. Es un error intentar calcular $suc(i_n)$.
- **funcion** *pred(x: T): T;* es el PREDecesor de *x* en la lista declarada: $pred(i_2) = i_1$, etc.. Es un error intentar calcular $pred(i_1)$.

Por ultimo, si a la declaracion de tipo ordenado le añadimos la palabra reservada **ciclico**

tipo *T* **es** *(i_1,i_2, ..., i_n)* **ciclico**;

obtenemos un *tipo enumerado ciclico*, que suprime, con respecto al tipo ordenado no ciclico, las limitaciones de *pred* y *suc* suponiendo que

$suc(i_n) = i_1$
$pred(i_1) = i_n$

La ordenacion es la misma que en los no ciclicos.

Ejemplos

Los tipos enumerados se utilizan para describir objetos cuyo rango de variacion es una coleccion de valores describible mediante identificadores.

Los tipos no ordenados son el caso mas sencillo:

tipo *palo* **es** {*oros, copas, bastos, espadas*};

tipo *fruta* **es** {*platano,manzana,pera,naranja*};

tipo *mueble* **es** {*mesa,silla,armario,percha,sofa*};

y se utilizan cuando no hay ninguna ordenacion implicita en el conjunto de valores (no tiene sentido preguntarse si *oros* < *copas* o *bastos* ≥ *espadas*). Se verifica

ord(oros) = 1
palo(2) = bastos
mueble(3) = percha
asc(copas,espadas) es la secuencia copas, bastos, espadas

Los tipos ordenados son adecuados cuando el orden es importante:

tipo *valor* **es** *(as, dos, tres, cuatro, cinco, seis, siete, ocho, nueve, sota, caballo, rey)*;

tipo *ciclo* **es** *(primero, segundo, doctorado)*;

En este caso, si pueden compararse los valores. Se tiene

as ≤ x para cualquier x: valor
pred(rey) = caballo
suc(cinco) = seis
suc(rey) produce un error

Y los ciclicos, cuando la estructura subyacente al conjunto de valores lo es:

tipo *dia* **es** *(lunes, martes, miercoles, jueves, viernes, sabado, domingo)* **ciclico**;

En este caso, es logico suponer que el SUCesor (o siguiente) del *Domingo* es el *Lunes*.

Sintaxis

```
declaracion_de_tipo =
  tipo identificador [es [nombre] tipo].

tipo = identificador
  | TIRA "("expresion_constante")"
  | tipo_enumerado
  | tipo_subrango
  | tipo_tabla
  | tipo_aplicacion
  | tipo_tupla
  | tipo_conjunto
  | tipo_fila .
```

Figura 42. Sintaxis de los tipos y sus declaraciones

```
tipo_enumerado = tipo_enumerado_no_ordenado
  | tipo_enumerado_ordenado
  | tipo_enumerado_ciclico

tipo_enumerado_no_ordenado = "{" lista_de_identificadores "}"

tipo_enumerado_ordenado = "(" lista_de_identificadores ")"

tipo_enumerado_ciclico = "(" lista_de_identificadores ")" ciclico
```

Figura 43. Sintaxis de los tipos enumerados.

Los tipos *entero*, *caracter* y *logico* como enumerados

Los tipos predefinidos *entero*, *caracter* y *logico*, aunque no son tipos enumerados, comparten algunas caracteristicas con ellos:

- El tipo *entero* podria considerarse declarado como

  tipo *entero* **es** *(...,-3,-2,-1,0,1,2,3,...);*

  donde los puntos significan que habria que escribir todos los enteros que admite el ordenador, y ordenados. Desde luego que esto contradice la sintaxis; ademas, la operacion *ord*, que es la identidad sobre los enteros, no se corresponde con el significado para tipos enumerados, y no existe la conversion de tipo *ENTERO*.

- El tipo *caracter* podria ser

  tipo *caracter* **es** *(... 'A' ... 'B' ... 'C' ...);*

  Incluyendo bien ordenados las letras y los numeros; el orden concreto depende del conjunto de caracteres. En este caso son aplicables todas las operaciones de los tipos enumerados, incluyendo *suc, pred, asc, desc, ord* y *caracter*. Dado que se supone la contiguidad de los digitos, esto permite calcular mediante *suc(c)* el caracter asociado al digito siguiente al representado en el caracter *c*, y el valor entero que representa un tal digito *c* mediante

  ord(C) - ord('0')

- El tipo *logico* se ajusta totalmente a la estructura de tipo enumerado

  tipo *logico* **es** *(falso,cierto);*

  añadiendo ademas sus propios operadores.

Subrangos

Los tipos subrango se utilizan para limitar el valor que puede tomar un objeto a un subintervalo de los valores de otro tipo.

tipo_subrango = "[" expresion_constante .. expresion_constante "]"

Figura 44. Sintaxis de un tipo subrango

Se dice que el *tipo base* de un tipo subrango es el tipo de las expresiones constantes que lo acotan. El tipo base ha de ser siempre entero, caracter, logico o enumerado.

Los valores que puede tomar un objeto de tipo subrango son los mismos que podria tomar si estuviese declarado con su tipo base, con la limitacion de que estos valores deben estar comprendidos entre los valores de las expresiones constantes que lo declaran.

Es posible utilizar una expresion de tipo subrango en cualquier ocasion en la que se requiera una expresion cuyo tipo sea el tipo base (en particular, en las asignaciones), y viceversa.

Ejemplos:

tipo *dia* **es** *[1..31]; --dias del mes; tipo base entero*

tipo *digito* **es** *['0'..'9']; -- tipo base caracter*

var *c: caracter; d: digito;*

c ← d; -- correcto aunque c y d no sean del mismo tipo

d ← c; -- tambien es correcto; si c no esta entre '0' y '9', se produce un error.

Es un error asignar a una variable de tipo subrango un valor de su tipo base no comprendido en el subrango. Pueden definirse varios tipos subrango a partir del mismo tipo base.

Cambio de fecha

El Algoritmo que se desarrolla a continuacion calcula el dia siguiente al de una fecha dada, incluyendo dia de la semana, dia del mes, mes y año. Los conceptos de tipo enumerado y tipo subrango permiten describir con precision los datos utilizados:

- Los dias de la semana se describen adecuadamente mediante

  tipo *dia_de_la_semana* **es** *(lunes, martes, miercoles, jueves, viernes, sabado, domingo)* **ciclico**;

- Igualmente, los meses:

  tipo *mes* **es**
  (enero, febrero, marzo, abril, mayo, junio,
  julio, agosto, septiembre, octubre, noviembre, diciembre) **ciclico**;

- En cuanto a los dias del mes:

  tipo *dia* **es** *[1..31];*

- Y los años

  tipo *año* **es** *[1..2000];*

El programa es trivial pero interesante para observar el funcionamiento de enumerados y subrangos. Se observara que datos erroneos (como 45 de Enero o año -7) son detectados automaticamente como errores gracias a las declaraciones efectuadas. En cuanto a la condicion trivial *es_el_ultimo_dia_del_mes*, se ha escrito como simplificacion, suponiendo que todos los meses tienen 31 dias.

```
programa Halla_la_siguiente_fecha es

  tipo dia_de_la_semana es (Lunes,Martes,Miercoles,Jueves,Viernes,Sabado,Domingo) ciclico;
  tipo mes es
    (Enero,Febrero,Marzo,Abril,Mayo,Junio,
    Julio,Agosto,Septiembre,Octubre,Noviembre,Diciembre) ciclico;
  tipo dia es [1..31];
  tipo año es [1..2000];

  var ds: dia_de_la_semana;
  var d: dia;
  var m: mes;
  var a: año;

  condicion es_el_ultimo_dia_del_mes haz
    -- para simplificar, suponemos que todos los meses tienen 31 dias; puede pensarse,
    -- como ejercicio, en cuales son las modificaciones necesarias para tratar
    -- los meses de 30, 29 y 28 dias, y la problematica de los años bisiestos
    vale d = 31;
  fin es_el_ultimo_dia_del_mes;

haz
  Escribe_linea "Que dia de la semana es?"; lee ds;
  Escribe_linea "Introduce la fecha de hoy:(Dia,Mes,Año)";
  lee d,m,a;

  -- Calculamos el siguiente dia de la semana.
  ds ← suc(ds);

  -- Calculamos la siguiente fecha
  si es_el_ultimo_dia_del_mes entonces
    si m = Diciembre entonces a ← a + 1; fin;
    d ← 1; m ← suc(m);
  sino
    d ← d + 1;
  fin si;

  -- Escribe la fecha de mañana
  Escribe "Mañana sera ",ds:1,", ",d:1," de ",m:1," de ",a:1,".";

fin programa;
```

Algoritmo 16. Halla la fecha siguiente a una dada.

Ejercicios

1. Extender el Algoritmo 16 como se indica en su listado.
2. Utilizando lectura de caracteres, pero no lectura de enteros, hacer un programa que lea numeros enteros y los escriba utilizando escritura de enteros y no de caracteres.

Otras instrucciones. Mas sobre entrada y salida

```
instruccion =
  instruccion_de_asignacion
  | instruccion_de_invocacion
  | instruccion_nada
  | instruccion_vale
  | instruccion_acaba
  | instruccion_produce
  | instruccion_sal
  | instruccion_si
  | instruccion_segun
  | instruccion_decide
  | instruccion_repite
  | instruccion_mientras
  | instruccion_itera
  | instruccion_para
  | instruccion_con
```

Figura 45. Instrucciones

Instruccion mientras

La instruccion **repite** ejecuta sus instrucciones subordinadas al menos una vez (y posiblemente varias, dependiendo del valor que tome la condicion). Esto no siempre es conveniente, por lo que debe recurrirse en ocasiones a esquemas del tipo

si *C* **entonces**
  repite
    I
  hastaque no *C;*
fin si;

Definimos la instruccion **mientras**

mientras *C* **haz**
  I
fin mientras;

como una abreviacion del esquema anterior.

Su forma general es

```
instruccion_mientras =
  mientras condicion haz
    instruccion
    {instruccion}
  fin [mientras];
```

Estilo: ademas del sugerido en la sintaxis,

mientras *condicion* **haz** *instruccion(es)* **fin**;

Figura 46. Sintaxis y estilo de la instruccion **mientras**

Se observara que es posible que las instrucciones subordinadas no se ejecuten en absoluto, con lo cual el efecto de la instruccion **mientras** puede ser nulo.

Instrucciones itera y sal

La forma pura de la iteracion se describe mediante la instruccion **itera**:

```
instruccion_itera
  itera
    instruccion
    {instruccion}
  fin [itera];
```

Estilo: ademas del sugerido en la sintaxis.

itera *instruccion(es)* **fin**;

Figura 47. Sintaxis y estilo de la instruccion **itera**

que tiene como efecto la ejecucion indefinidamente repetida de las instrucciones subordinadas. Suele utilizarse en conjuncion con la instruccion **sal**:

instruccion_sal = **sal** *[identificador]* [**cuando** *condicion];*

Figura 48. Sintaxis de la instruccion **sal**

en la forma

```
itera
  instruccion(es)
sal cuando condicion;
  instruccion(es)
fin itera;
```

Figura 49. Esquema normal de utilizacion de la instruccion **itera**

El efecto de la instruccion **sal** es el de terminar la ejecucion de la instruccion **itera** y continuar con la primera instruccion que sigue a esta. Un caso tipico de utilizacion es el siguiente:

En un programa deseamos leer el dia del mes, que sabemos estara comprendido en el rango [1..31]. Deseamos, ademas, *recuperar errores*, esto es, prever los casos en que el usuario del programa introduce equivocadamente un numero no valido, informarle de su error, y darle la oportunidad de corregirlo. Para ello utilizamos la siguiente codificacion:

```
itera
  escribe_linea "Escribe el dia del mes:";
  lee n;
sal cuando n ≥ 1 y n ≤ 31;
  escribe_linea "Error: ha de estar entre 1 y 31.";
fin itera;
```

de la cual puede ser la siguiente una utilizacion tipica

```
Escribe el dia del mes:
32
Error: ha de estar entre 1 y 31.
Escribe el dia del mes:
31
 ... (el programa continua)
```

La instruccion **sal** es mas general: incorpora en su sintaxis la posibilidad de mencionar explicitamente de que iteracion se desea **salir**; aunque es poco frecuente, a veces interesa terminar no con la iteracion que inmediatamente engloba a **sal**, sino con alguna mas externa. Esto puede hacerse utilizando un *nombre de iteracion*, en la forma de un identificador seguido de dos puntos que precede a la instruccion cuya ejecucion se quiere terminar, y mencionando este identificador en la instruccion **sal**:

```
Iteracion_externa:
 itera
  ...
  Iteracion_interna:
   itera
    ...
    sal Iteracion_externa cuando C;
    ...
   fin itera;
 fin itera;
 -- la instruccion sal continua la ejecucion en este punto
```

Tambien es posible omitir "**cuando** condicion" en la instruccion **sal**, en cuyo caso se supone "**sal cuando** *cierto*". De este modo, es equivalente escribir

```
sal cuando C;
```

y

```
si C entonces sal; fin;
```

Pueden utilizarse tantas instrucciones **sal** dentro de una iteracion como se desee, aunque, por razones de claridad, debe procurarse restringirse al esquema presentado. La instruccion **sal** sirve tambien para terminar la ejecucion de una instruccion **para**, **repite** o **mientras**; por su parte, estas admiten tambien un nombre de bucle.

La ejecucion de una iteracion puede tambien terminarse mediante la de una instruccion **vale** o **acaba**, o suspenderse con una **produce**.

Algunas equivalencias

La combinacion **itera-sal** es mas general que las otras formas de iteracion: si en la forma

```
itera
  i1
sal cuando C;
  i2
fin itera;
```

suponemos i_1 = **nada;**, obtenemos

```
itera
  nada;
sal cuando C;
  i2
fin itera;
```

que equivale evidentemente a

```
mientras no C haz
  i2
fin mientras;
```

Igualmente, si es i_2 lo que substituimos por **nada**;, obtenemos

```
itera
  i1
sal cuando C;
  nada;
fin itera;
```

que en este caso equivale a

```
repite
  i1
hastaque C;
```

De modo que las instrucciones **mientras** y **repite** pueden considerarse formas especiales de la **itera**.

Instruccion segun

```
instruccion_segun =
  segun expresion es
    cuando lista_segun =>
      instruccion
      {instruccion}
    {cuando lista_segun =>
      instruccion
      {instruccion}}
    [cuando otros =>
      instruccion
      {instruccion}]
  fin [segun];

lista_segun = valor_o_rango { "|" valor_o_rango }

valor_o_rango = expresion_constante [ .. expresion_constante ]
```

Estilo: si la(s) instruccion(es) caben en la misma linea que **cuando**, puede hacerse

```
  cuando lista_segun => instruccion(es);
```

o

```
  cuando otros => instruccion(es);
```

***Figura 50. Sintaxis y estilo de la instruccion* segun**

Si en una instruccion **decide** todas las condiciones se refieren al valor de una misma expresion o variable

```
decide
  cuando c = '.' o c = '*' => I1;
  cuando c = 'a' o c = 'A' => I2;
  cuando otros => I3;
fin decide;
```

puede abreviarse, ganando en claridad, por

```
segun c es
  cuando '.' | '*' => I1;
  cuando 'a' | 'A' => I2;
  cuando otros => I3;
fin segun;
```

haciendo una especie de "factor comun" de la expresion en cuestion. Ademas, puede utilizarse la notacion

valor$_1$..*valor*$_2$

para expresar el conjunto de valores comprendidos entre *valor*$_1$ y *valor*$_2$:

'0'..'9'

substituye a

'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Como en la instruccion **decide**, se produce un error si falta **"cuando otros"** y la expresion no coincide con ninguno de los valores expresados; se notara que, en esta instruccion, han de ser *expresiones constantes* (esto es, expresiones que se evaluen como e involucren solo constantes), todas ellas distintas. Ademas, el tipo de la expresion solo puede ser *caracter, entero, logico*, enumerado o subrango.

La diferencia entre los ordinales del minimo y el maximo valor del conjunto de las expresiones constantes debera ser reducida; en caso contrario, se preferira la instruccion **decide** (o la **si**): no se escribira

```
segun n es
  cuando 1 => I1;
  cuando 100 => I2;
  cuando 1000 => I3;
fin segun;
```

sino

```
decide
  cuando n = 1 => I1;
  cuando n = 100 => I2;
  cuando n = 1000 => I3;
fin decide;
```

y se preferira

```
decide
  cuando k ≥ 1 y k ≤ 100 => I1;
  cuando k ≥ 101 y k ≤ 200 => I2;
fin decide;
```

a

```
segun k es
  cuando 1..100 => I1;
  cuando 101..200 => I2;
fin segun;
```

Ejemplo: Suponiendo declarado el tipo *dia* (de la semana), y una variable *d* de este tipo,

```
segun d es
  cuando lunes..miercoles => instruccion1;
  cuando viernes | domingo => instruccion2;
  cuando otros => instruccion_3;
fin segun;
```

Formatos de entrada y salida

Normalmente, las instrucciones *escribe* y *escribe_linea* utilizan automaticamente un determinado numero de espacios para escribir cada tipo de datos: el minimo necesario para todos los tipos, excepto para los reales (como se vera a continuacion). Esto quiere decir que una instruccion como

escribe_linea "Hay ",n," elementos.";

para *n = 3* escribira

```
Hay 3 elementos.
```

y para *n = 12345* escribira

```
Hay 12345 elementos.
```

ajustando el numero de espacios a la magnitud del numero. A veces interesa evitar este ajuste automatico (por ejemplo, para producir listados bien encolumnados). Es posible determinar el numero de espacios utilizados en la escritura de cualquier valor sin mas que especificar, a continuacion del valor y separado por el simbolo ':', el numero de espacios que se desea que ocupe.

Ejemplo:

escribe ' El numero de As es: ',n:3;

para N = 5 produce

```
El numero de As es:   5
```

Si el numero de espacios especificados no es suficiente para acomodar el valor a escribir, automaticamente se toman mas espacios hasta poder escribir la totalidad del valor: la instruccion anterior, con n = 1234, produciria

```
El numero de As es: 1234
```

Si el numero de espacios especificado en el formato es superior al necesario, se escriben blancos antes del valor.

El caso de los numeros reales es algo mas complejo: se considera que la expresion minima de un numero real consta de

un espacio en blanco
el signo (opcional) del numero
un digito antes del punto decimal
el punto decimal
un digito despues del punto decimal
la E de exponente
el signo del exponente
dos digitos para el exponente

lo cual totaliza 9 espacios. Si el formato de escritura para un valor real es inferior a 9, se utilizan 9 espacios, y se representa el numero utilizando su expresion minima. A medida que el formato especifica un numero mayor de espacios, crece el numero de digitos que aparecen despues del punto decimal, hasta llegar al maximo de digitos que permite la version. El espacio adicional que especifique el formato se utiliza en forma de espacios en blanco que preceden al numero.

Entrada y salida interactiva

El efecto asignado a las acciones predefinidas de entrada y salida *lee*, *lee_linea*, *escribe* y *escribe_linea* obliga a programar lo que se desea que sean dialogos interactivos entre el programa y el usuario de modos no obvios. La transmision de datos sobre la pantalla se realiza linea a linea: esto quiere decir que varias instrucciones *escribe* "acumulan" sus resultados hasta la ejecucion de una *escribe_linea*, y que varias *lee* utilizan la misma linea de entrada, si hay datos y no media alguna *lee_linea*. Por ejemplo, si programamos

lee m,n;

y la linea de entrada contiene

2 4 6

M y *n* tomaran por valor, respectivamente. 2 y 4. pero el numero 6 quedara pendiente de ser leido; una instruccion siguiente

lee p;

asignara 6 a *P*. Para evitar esta acumulacion, se utiliza a veces la instruccion *lee_linea*; pero si escribimos en vez de *lee*

lee_linea m,n;

para evitar leer datos adicionales extraños contenidos en la linea. el efecto es que, despues de leer correctamente *m* y *n*, el ordenador pide una nueva linea de datos (es lo que hace *lee_linea*: despues de leer las variables designadas, pide una nueva linea de datos), que seguramente no estamos en condiciones de proporcionar en ese momento. Para evitar anticipaciones, la unica forma es hacer que a *la segunda* instruccion *lee* le preceda una *lee_linea* vacia:

lee_linea; lee p;

que ignorara el resto de la linea anterior, y obtendra de la nueva el dato que buscamos.

En general. el esquema de dialogo interactivo puede ser el siguiente:

```
escribe_linea linea1;
lee datos1;
escribe_linea linea2;
lee_linea; lee datos2;
escribe_linea linea3;
lee_linea; lee datos3;
 ...
```

Figura 51. Esquema general de dialogo interactivo

Calculo del Maximo Comun Divisor

El siguiente programa calcula el Maximo Comun Divisor de dos numeros naturales, basandose en las reglas

$mcd(a,b) = mcd(a-b,b)$
$mcd(a,a) = a$

y es interesante como ejemplo de las instrucciones explicadas.

```
programa Maximo_Comun_Divisor es
  var a,b: entero;
haz
  itera
    itera
      escribe_linea "Escribe dos numeros enteros, o 0 0 para terminar:";
      lee a,b;
    sal cuando (a > 0 y b > 0) o (a = 0 y b = 0);
      escribe_linea "Han de ser positivos.";
    fin itera;
  sal cuando a = 0 y b = 0;
    escribe "El M.C.D. de ",a," y ",b," es: ";
    mientras a ≠ b haz
      -- a > b ⇒ mcd(a,b) = mcd(a-b,b)
      -- b > a ⇒ mcd(a,b) = mcd(a,b-a)
      si a > b entonces a ← a - b; sino b ← b - a; fin;
    fin mientras;
    -- a = b ⇒ mcd(a,b) = a = b
    escribe_linea a,'.';
  fin itera;
fin programa;
```

Algoritmo 17. Calculo del Maximo Comun Divisor

Ejercicio

Revisar todos los programas escritos hasta ahora, para ver cuales ganan en claridad reescribiendolos mediante las nuevas instrucciones. Observar si alguno resuelve enunciados mas generales al reescribirlo (por ejemplo, al substituir **repite** por **mientras**).

Conjuntos

Los conjuntos como abreviacion de expresiones

Una condicion como

c = *'a'* **o** *c* = *'e'* **o** *c* = *'i'* **o** *c* = *'e'* **o** *c* = *'u'*

puede cambiarse, con el mismo efecto, por

c **en** {*'a','b','c','d','e'*}

utilizando un *conjunto* y el *operador de pertenencia* "**en**". Pueden escribirse conjuntos de tipo *entero, caracter, logico*, enumerado o subrango (con las limitaciones que se estudiaran a continuacion) encerrando sus elementos entre llaves; los elementos pueden ser (expresiones) constantes, variables o expresiones, y el significado es el usual en matematicas elementales.

---

conjunto = "{" *expresion* {, *expresion*} "}"

Figura 52. Sintaxis de un conjunto

Tipos conjunto y operaciones

Tambien pueden definirse objetos que contengan tales conjuntos, directamente, o mediante la definicion de un tipo

---

tipo_conjunto = **conjunto de** *tipo_base*

tipo_base = *identificador* | *tipo_enumerado* | *tipo_subrango*

***Figura 53. Sintaxis de un tipo* conjunto**

El *tipo base* debe ser *logico, caracter*, enumerado, o un subrango de estos o de *entero*; sus ordinales minimo y maximo estan sujetos a limitaciones dependientes de version (generalmente, el minimo no puede ser menor que cero, y el maximo que el mayor ordinal de un caracter).

Asi, puede definirse

tipo *mueble* **es** {*mesa,silla,armario,sofa,cama*};
tipo *mobiliaro* **es conjunto de** *mueble;*
var *mu: mueble; mo: mobiliario;*

y hacer

mo ← {*mesa,sofa*};
mu ← *armario;*
mo ← {*mu,silla*};

Pueden definirse constantes que sean conjuntos, aunque en este caso los elementos deberan ser (expresiones) constantes:

const *mo1* = {*sofa.cama*};

Las operaciones aplicables a los conjuntos son:

- La asignacion y la comparacion por (des)igualdad.
- La *pertenencia*, denotada por el operador **en**, de la que se han visto ya ejemplos, y cuyo resultado es de tipo *logico: cierto* si y solo si el elemento pertenece al conjunto.
- La *inclusion* conjuntista (no estricta), que permite determinar si un conjunto esta contenido en otro o, lo que es lo mismo, si cada elemento de un conjunto pertenece tambien a otro.

  $A \leq B$ ⇔ **no existe** x **en** A **talque no** *(*x **en** B*)*
  $A \geq B$ ⇔ **no existe** x **en** B **talque no** *(*x **en** A*)*

  Se denota mediante los operadores "≤" y "≥":

  {silla,mesa} ≤ *{silla,mesa,sofa}*
  {sofa,mesa} ≥ *{sofa}*

  y su resultado es tambien de tipo logico. No hay operadores predefinidos para evaluar inclusiones estrictas, pero puede recurrirse a

  A incluido estrictamente en B ⇔ $A \leq B$ y $A \neq B$

  y simetricamente.

- La *union* (operador "+"), *interseccion* (operador "*") y *diferencia* (operador "-") de conjuntos, con el significado matematico tradicional:

  $A + B = \{x \mid x$ **en** A **o** x **en** $B\}$
  $A * B = \{x \mid x$ **en** A **y** x **en** $B\}$
  $A - B = \{x \mid x$ **en** A **y no** *(*x **en** B*)*$\}$

  Por ejemplo:

  {mesa,silla} + *{sofa}* = *{mesa,silla,sofa}* -- *union*

  {1,3,5,7} + *{1,2,3,4}* = *{1,2,3,4,5,7}* -- *union*

  {1,2,3,4,5} * *{1,3,5,7}* = *{1,3,5}* -- *interseccion*

  {1,2,3,4,5} - *{2,4,6,8}* = *{1,3,5}* -- *diferencia:*
  -- elementos del primer conjunto que no estan en el segundo

  {'a','e','i','o','u'} - *{'z'}* = *{'a','e','i','o','u'}* -- *diferencia*

Para respetar las reglas de compatibilidad de tipos, los conjuntos con los que se opera deben tener el mismo tipo base (o bien tipos base que sean subrangos del mismo tipo base). Entre los conjuntos constantes se cuenta el *conjunto vacio*, que tiene cualquier tipo (conjunto), y se denota {}.

Algunos ejemplos

Para determinar si un caracter es una letra mayuscula, puede ser muy util una constante

```
const mayusculas =
  {'A','B','C','D','E','F','G','H','I','J','K','L','M',
  'N','Ñ','O','P','Q','R','S','T','U','V','W','X','Y','Z'};
```

que permitira preguntas del tipo

si *c* **en** *mayusculas* **entonces** ...

Generacion de numeros primos mediante la criba de Eratostenes y conjuntos:: Un metodo bastante conocido para generar una lista de numeros primos es el siguiente: se parte de la lista de los numeros enteros:

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

Despues de tachar el numero 1 (que no es primo), se mira el primer numero no tachado (2). Este numero es primo (se apunta en la lista); se tachan todos los multiplos de 2 a partir de su cuadrado (4):

	2	3	5	7	9
11		13	15	17	19
21		23	25	27	29
31		33	35	37	39
41		43	45	47	49
51		53	55	57	59
61		63	65	67	69
71		73	75	77	79
81		83	85	87	89
91		93	95	97	99

El siguiente numero es el 3 (primo, que se apunta en la lista). Se tachan ahora los multiplos de 3 empezando por su cuadrado (9):

	2	3	5	7	
11		13		17	19
		23	25		29
31			35	37	
41		43		47	49
		53	55		59
61			65	67	
71		73		77	79
		83	85		89
91			95	97	

El siguiente numero (5) es primo ... Se repite asi el proceso, con lo que se obtiene al final una tabla de los primos, al haber eliminado los que no lo son:

	2	3	5	7	
11		13		17	19
		23			29
31				37	
41		43		47	
		53			59
61				67	
71		73			79
		83			89
				97	

El siguiente algoritmo implementa este proceso; la lista de naturales se representa mediante una variable de tipo conjunto. Se observara que dada la ausencia de una operacion predefinida para calcular complementos, es necesario construir el conjunto de los naturales "a mano" (instruccion **para**).

```
programa Numeros_primos es
  const max = 200;
  var T: conjunto de [1..max]; -- Los naturales
  var i,p,q: entero;
haz
  T ← {};
  -- Llena T con los naturales:
  para i en asc(1,max) haz T ← T + {i}; fin;
  -- Quita el 1.
  T ← T - {1};
  p ← 1;
  repite
    -- Busca un p en T, ...
    mientras p ≤ max y entonces no (p en T) haz
      p ← p + 1;
    fin mientras;
    -- (solo si no te has pasado)
    si p ≤ max entonces
      -- ... que es primo.
      escribe_linea p;
      -- Calcula su cuadrado; eliminalo ...
      q ← p * p;
      mientras q ≤ max haz
        T ← T - {q};
        -- ... con todos sus multiplos a partir de ahi.
        q ← q + p;
      fin mientras;
      -- Prueba con el siguiente natural.
      p ← p + 1;
    fin si;
  hastaque p > max;
fin programa;
```

Algoritmo 18. Generacion de numeros primos la Criba de Eratostenes

Tablas o aplicaciones

Familias indexadas de variables

Para hallar el mínimo *min* de dos variables n_1 y n_2 puede hacerse

si $n_1 < n_2$ **entonces** $min \leftarrow n_1$; **sino** $min \leftarrow n_2$; **fin**;

Si las variables son tres, n_1, n_2 *y* n_3, es mas complicado:

```
decide
  cuando n1 ≤ n2 y n1 ≤ n3 ⇒ min ← n1;
  cuando n2 ≤ n1 y n2 ≤ n3 ⇒ min ← n2;
  cuando n3 ≤ n1 y n3 ≤ n2 ⇒ min ← n3;
fin decide;
```

creciendo la complejidad con el numero de variables. Si de algun modo pudiese hablarse de n_i, siendo *i* una *variable* (o una expresion) entre 1 y *k*, se podria escribir un esquema general:

```
min ← n1;
para i en asc(2.k) haz
  si ni < min entonces min ← ni; fin;
fin para;
```

considerando *n* como una *familia indexada de variables.*

Esto no es posible en UBL (entre otras cosas, debido que no es posible escribir subindices y a la ambiguedad que resultaria de intentar diferenciar n_i y el identificador *ni*). Aunque con otra notacion, el tipo **tabla** viene a resolver este problema: podemos declarar

var *n:* **tabla** *[1..k]* **de** *entero;*

(donde *k* es una constante) y referirnos a cada *elemento* o *componente* de la familia utilizando una expresion como *indice*:

```
n[1] -- antes n1
n[3] -- n3
n[i] -- ni
n[i + j] -- el indice es una expresion
n[n[1]] -- porque no?
```

En este caso, *n* es un *objeto estructurado*, compuesto de variables individuales accesibles mediante el indexado o *seleccion.*

n

3	-1	4	6	0	0	5
n[1]	*n[2]*	*n[3]*	*n[4]*	*n[5]*	*n[6]*	*n[7]*

Figura 54. ***Una*** **tabla** ***de 7 enteros, contemplada como*** **tabla** ***y como familia de variables subindiciadas***

Naturalmente, tambien se podria haber utilizado un tipo

tipo *numeros* **es tabla** *[1..k]* **de** *entero;*

que hubiese permitido declarar varios objetos similares.

Aplicaciones

Para pasar un caracter "a mayusculas" (es decir, transformar los caracteres que sean minusculas en sus correspondientes mayusculas) necesitamos algun mecanismo para implementar la aplicacion

```
          Mayuscula
'a'   ----------> 'A'
'b'   ----------> 'B'
'c'   ----------> 'C'
'd'   ----------> 'D'
          ....
'z'   ----------> 'Z'
```

Ademas del metodo obvio pero pesado de utilizar una funcion

```
funcion mayuscula(c: caracter): caracter haz
  segun c es
    cuando 'a' => vale 'A';
    ...
    cuando 'z' => vale 'Z';
  fin segun;
fin mayuscula;
```

o del mas sofisticado con dos tiras

```
min ← "abcdefghijklmnñopqrstuvwxyz";
may ← "ABCDEFGHIJKLMNÑOPQRSTUVWXYZ";
si existe i en asc(1,long(min)) talque min[i] = c entonces
  c ← may[i];
fin si;
```

puede programarse mediante un tipo **aplicacion**:

se declara

```
var mayuscula: aplicacion caracter => caracter;
```

y se inicializa

```
mayuscula['a'] ← 'A';
mayuscula['b'] ← 'B';
mayuscula['c'] ← 'C';
  ...
mayuscula['z'] ← 'Z';
  ...
c ← mayuscula[c];
```

que reduce el calculo al contener la aplicacion como objeto siempre calculado en memoria.

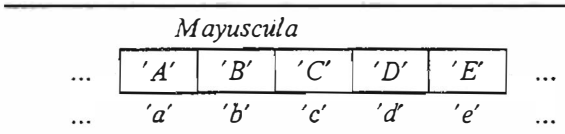


Figura 55. Fragmento de una **aplicacion** *de los caracteres en los caracteres.*

Tabulacion

En un determinado trabajo, cada dia

```
tipo dia es (lunes,martes,miercoles,jueves,viernes,sabado,domingo) ciclico;
```

se sale a una hora distinta. Esa información está contenida en una tabla horaria

```
tipo hora es [1..24];
tipo laborable es [lunes..viernes];
var hora_de_salida: tabla laborable de hora;
```

en la forma

```
hora_de_salida [lunes]     ← 7;
hora_de_salida [martes]    ← 5;
hora_de_salida [miercoles] ← 7;
hora_de_salida [jueves]    ← 3;
hora_de_salida [viernes]   ← 6;
```

de modo que puede utilizarse, dado

```
var l: laborable;
```

la expresion

```
hora_de_salida[l]
```

para calcular la hora de salida.

Hora_de_salida

Lunes	*7*
Martes	*5*
Miercoles	*7*
Jueves	*3*
Viernes	*6*

Figura 56. Tabla de los horarios de salida del trabajo en dias laborables

Vectores

Deseamos construir un tipo que nos permita representar coordenadas en el espacio tridimensional:

```
(0,0,0)
(1,-1,0)
```

Declaramos

```
tipo punto es tabla [1..3] de real;
```

o bien, y equivalentemente,

```
tipo punto es aplicacion [1..3] ⇒ real;
```

y luego, objetos de ese tipo

```
var p,q,r: punto;
```

Podemos asignar una coordenada a otra:

```
p ← q;
```

hallar las proyecciones de una coordenada

```
p[1] -- puede leerse p sub 1
q[3] -- q sub 3
p[i] -- p sub i: altura, anchura o profundidad, segun i
```

o encontrar el producto escalar de los vectores asociados a *p* y *q*:

```
prod ← 0;
para i en asc(1,3) haz prod ← prod + p[i]*q[i]; fin;
```

Los tipos tabla y aplicacion

tipo_tabla = **tabla** *tipo_indice* {,*tipo_indice*} **de** *tipo_elemento*

tipo_indice = *identificador* | *tipo_enumerado* | *tipo_subrango*

tipo_elemento = *tipo*

Figura 57. Sintaxis de un tipo **tabla**

tipo_aplicacion = **aplicacion** *tipo_origen* {,*tipo_origen*} ⇒ *tipo_imagen*

tipo_origen = *identificador* | *tipo_enumerado* | *tipo_subrango*

tipo_imagen = *tipo*

Figura 58. Sintaxis de un tipo **aplicacion**

tabla y **aplicacion** son sinonimos; solo cambia la sintaxis de la declaracion de tipo, y la forma de conceptualizarlo, como se ha visto en los ejemplos. Un objeto de tipo **tabla** o **aplicacion** representa una *familia indexada* de objetos, que son accesibles individualmente mediante la operacion de *seleccion.*

variable = *identificador* {*selector*}

selector = ↑
| "[" *expresion* {,*expresion*} "]"
| . *identificador*

Figura 59. Sintaxis de una variable con sus posibles selectores:: "↑" y ". identificador" se estudiaran mas adelante

Hay tantos *elementos* o *componentes* como valores en el *tipo_indice* (o *tipo_origen*), que ha de ser *caracter, logico*, enumerado o subrango; cada elemento se selecciona añadiendo al nombre del objeto el valor del indice encerrado entre corchetes.

Abreviaciones: Se consideran equivalentes las construcciones

tabla T_1 **de tabla** T_2 **de ... tabla** T_n **de** T

y

tabla T_1, T_2, ... ,T_n **de** T

asi como

aplicacion O_1 ⇒ **aplicacion** O_2 ⇒ ... **aplicacion** O_n ⇒ I
I

y

aplicacion O_1 O_2 ... O_n $\Rightarrow$ *I*

y tambien las aplicaciones y las tablas entre si; igualmente,

v[i1][i2]...[in]

es intercambiable por

v[i1,i2,...,in]

con el mismo sentido.

Operaciones

Las operaciones aplicables a los objetos de estos tipos son:

- La asignacion.
- La comparacion por igualdad o por desigualdad.
- La *seleccion.* Esta es una operacion distinta de las estudiadas hasta el momento, en el sentido de que no produce un valor, sino un nombre (esto es, una [sub]variable). Dada una tabla

  var *T:* **tabla** *[1..10]* **de** *entero*

  los elementos *T[2]* o *T[3]* (al igual que la tabla entera *T*) pueden usarse, a todos los efectos, como variables: puede asignarseles valores o pasarlos como parametro (variable o no):

  ```
  T[2] ← T[3];
  T[i] ← T[j];
  lee T[k],T[i-k];
  ```

  Ello introduce nuevos problemas de alias: *T[i]* y *T[j]* pueden ser o no el mismo objeto, dependiendo del valor de *i* y *j*.

Variaciones sobre un mismo problema

El problema es el siguiente

> Dada una serie de numeros, escribir solo los que superen la media de todos ellos.

Y admite varias interpretaciones. Algunas de las posibles son: la cantidad de numeros es fija, o variable; y, en ese caso, viene determinada por un marcador (p. ej., el numero 0) al final de la serie, o indicada al principio en un numero adicional.

Cantidad fija de numeros: por ejemplo, 10 numeros. El problema no puede resolverse con la aproximacion utilizada en casos anteriores, ya que necesitamos "pasar dos veces" por cada numero: la primera, para calcular la media, y la segunda, para ver si la excede. Utilizaremos una **tabla** para almacenar los numeros.

```
const max = 10;
var N: tabla [1..max] de entero;
```

La constante *max* se ha declarado para hacer el programa mas general: en vez de 10 se escribe siempre *max*, y asi, si el enunciado cambia hacia otra cantidad distinta de 10, basta con cambiar la constante. Por lo demas, el programa es trivial.

```
programa Mayores_que_su_media_version_1 es
  const max = 10; -- cantidad de numeros
  var N: tabla [1..max] de entero;
  var i, media: entero;
haz
  media ← 0; -- lee y calcula la media a la vez
  para i en asc(1,max) haz
    lee N[i]; media ← media + N[i];
  fin para;
  media ← media div max;
  -- escribe los mayores que la media
  para i en asc(1,max) talque N[i] > media haz
    escribe_linea N[i];
  fin para;
fin programa;
```

Algoritmo 19. Filtra entre 10 numeros los mayores que su media

Cantidad variable de numeros: En este caso, el mecanismo que suele utilizarse es suponer un limite superior razonable (p. ej., 100) a la cantidad de numeros, y declarar una tabla de ese tamaño:

```
const max = 100;
var N: tabla [1..max] de entero;
```

Tambien se declara una variable para contener la cantidad

```
var c: entero;
```

y se trabaja como si N estuviera declarado con un indice igual a $[1..c]$ (cosa que el lenguaje no permite directamente, ya que los limites de un tipo subrango deben ser constantes). Con ello se consigue el efecto de *dimensiones ajustables*.

Si la serie de numeros esta terminada por un cero, el programa sera

```
programa Mayores_que_su_media_version_2 es
  const max = 10; -- maximo de numeros
  var N: tabla [1..max] de entero;
  var c, i, media: entero;
haz
  c ← 1; -- cantidad de numeros en la serie + 1
  media ← 0; -- lee y calcula la media a la vez
  itera -- en vez de PARA, ya que desconocemos C
    lee N[c];
  sal cuando N[c] = 0;
    media ← media + N[c];
    c ← c + 1;
  fin itera;
  media ← media div (c - 1);
  -- escribe los mayores que la media
  para i en asc(1,c-1) talque N[i] > media haz
    escribe_linea N[i];
  fin para;
fin programa;
```

Algoritmo 20. Filtra entre una serie de numeros terminada por 0 los mayores que su media

Y si el primer numero no pertenece a la serie, sino que indica cuantos le siguen,

```
programa Mayores_que_su_media_version_3 es
  const max = 10; -- maxima cantidad de numeros
  var N: tabla [1..max] de entero;
  var c, i, media: entero;
haz
  lee c; -- en este caso, lee C directamente
  media ← 0; -- lee y calcula la media a la vez
  para i en asc(1,c) haz
    lee N[i]; media ← media + N[i];
  fin para;
  media ← media div c;
  -- escribe los mayores que la media
  para i en asc(1,c) talque N[i] > media haz
    escribe_linea N[i];
  fin para;
fin programa;
```

Algoritmo 21. Filtra entre una serie de C numeros los mayores que su media

En este caso el esquema es practicamente igual al de la Version 1.

Cuando tabla, y cuando aplicacion?

tabla y **aplicacion** son sinonimos para el interprete. Suelen declararse **aplicaciones** cuando el uso que se hace del objeto se reduce practicamente a la operacion de seleccion, y/o cuando conceptualmente sea conveniente; en cambio, las **tablas** suelen utilizarse como "depositos" de objetos: muchas veces, con caracter *ajustable*, como en los anteriores ejemplos, y otras, de modos mas complejos. Como en el siguiente caso.

Inversion de una serie de numeros

Disponemos de una serie de numeros enteros terminada por un cero, y deseamos reescribirla invertida, esto es: dado

1 4 2 5 8 0

debemos escribir

8 5 2 4 1

Para ello, suponemos un limite superior a la longitud de la serie, y utilizamos una **tabla** como deposito lugar de la inversion, con ayuda de una variable que hace de señal o *apuntador*:

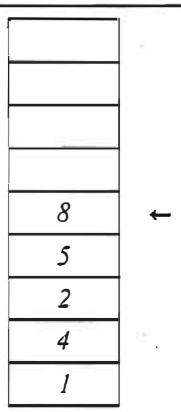


Figura 60. Utilizacion combinada de una **tabla** *y un apuntador*

La señal, representada por una flecha en la figura, es una variable cuyo valor es el indice del ultimo elemento leido. Cada vez que se lee un nuevo elemento, se corre "hacia arriba" la señal, y se inserta

el elemento en la tabla. Para obtener la serie en orden inverso, basta con seguir el recorrido descendente de la señal.

```
programa Da_la_vuelta es
  const max = 100;
  var T: tabla [1..max] de entero;
  var f: [0..max]; -- la flecha
  var n: entero;
haz
  f ← 0; -- de momento, ningun numero
  itera
    lee n;
  sal cuando n = 0;
    f ← f + 1;
    T[f] ← n;
  fin itera;
  para n en desc(f,1) haz
    escribe_linea T[n];
  fin para;
fin programa;
```

Algoritmo 22. Invierte una serie de numeros

Matrices

Asi como a las tablas o aplicaciones se les llama a veces *vectores*, las tablas de tablas (o aplicaciones, etc.) suelen utilizarse para representar, y tomar el nombre de, *matrices*.

Una matriz cuadrada de 3 x 3 puede ser declarada como

```
var M: tabla [1..3],[1..3] de real;
```

El elemento $M_{i,j}$ se denotara por *M[i,j]* o *M[i][j]*.

Las matrices pueden ser bidimensionales, tridimensionales o, en general, de cuaquier numero de dimensiones, segun las necesidades del programa. Si se utilizan menos subindices de los posibles (si en el ejemplo anterior hablamos de *M[I]*) se obtiene una *submatriz* de la matriz total, que generalmente representa una fila o columna de la abstraccion dada.

```
var M: tabla [1..3],[1..4] de entero;
```

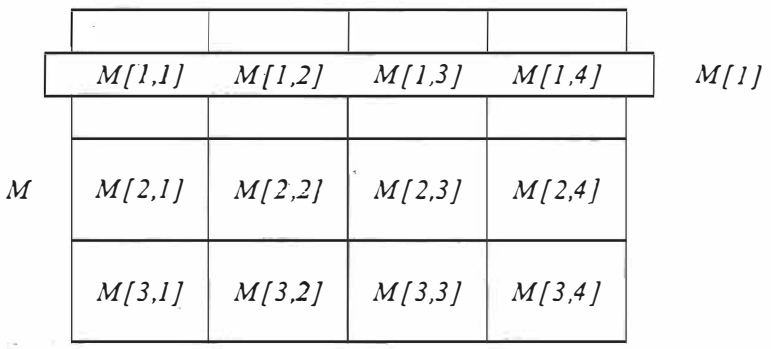


Figura 61. Una matriz con una submatriz: M es la matriz completa. Las lineas de puntos delimitan la submatriz *M[1]*, que a su vez es una tabla, con componentes *M[1][1] = M[1,1]*, *M[1][2]*, *M[1][3]* y *M[1][4]*, y tiene tipo **tabla** *[1..4]* **de** *entero*.

Un ejemplo de tratamiento de matrices: generacion de Cuadrados Magicos

Llamaremos cuadrado magico a una matriz cuadrada de numeros. de lado impar. tal que coincidan las sumas de los valores de sus filas. columnas y diagonal descendente:

Ejemplo:

8	*21*	*14*	*2*	*20*	*65*
25	*13*	*1*	*19*	*7*	*65*
12	*5*	*18*	*6*	*24*	*65*
4	*17*	*10*	*23*	*11*	*65*
16	*9*	*22*	*15*	*3*	*65*
65	*65*	*65*	*65*	*65*	*65*

Figura 62. Ejemplo de Cuadrado Magico de lado 5

El siguiente programa, dado un numero entero impar *n*, construye e imprime un cuadrado magico de tamaño *n*n*. Para ello, utiliza las siguientes reglas:

1. En todo momento se supone que los bordes del cuadrado estan identificados. Dicho de otro modo, convenimos en hablar como si, por ejemplo, a la ultima casilla de la primera fila le siguiera la primera casilla de la misma fila por la derecha, y la ultima de la ultima fila por arriba; y asi para cada casilla. Con ello conseguimos que toda casilla tenga "vecinos" en cualquier direccion.

2. Para llenar el cuadrado, se escriben sucesivamente los *n*n* primeros numeros enteros en cada una de las casillas. Se empieza por la situada encima de la central, y se sigue en direccion diagonal ascendente (en el sentido de la regla 1); si se llega a una casilla ocupada, se "rebota", realizando un paso en direccion diagonal descendente.

El programa es tambien un buen ejemplo de diseño descendente y de escritura con formatos.

```
programa Cuadrado_magico es

  const max = 25; -- tamaño maximo del cuadrado magico

  tipo cuadrado es tabla [1..max],[1..max] de entero;

  var i,j: [0..max+1]; n: [1..max];
  var c: cuadrado; k: entero;

  accion inicializa haz
    escribe_linea "Escribe la dimension del cuadrado:";
    itera
      lee n;
    sal cuando n ≥ 3 y n ≤ max y impar(n);
      escribe_linea
        "Valor erroneo. Ha de ser impar y estar entre 3 y ",max,". Escribe otro valor:";
    fin itera;
    para i en asc(1,n) haz para j en asc(1,n) haz c[i,j] ← 0; fin; fin;
    k ← 1; j ← n div 2; i ← (n + 1) div 2;
  fin inicializa;

  accion pon_un_numero haz c[i,j] ← k; k ← k + 1; fin;

  accion siguiente_casilla haz
    i ← i + 1; si i > n entonces i ← 1; fin;
    j ← j - 1; si j < 1 entonces j ← n; fin;
  fin siguiente_casilla;

  condicion esta_ocupada haz vale c[i,j] ≠ 0; fin;

  accion rebota haz
    i ← i + 1; si i > n entonces i ← 1; fin;
    j ← j + 1; si j > n entonces j ← 1; fin;
  fin rebota;

  condicion esta_lleno haz vale k > n * n; fin;

  accion imprime_resultados haz
    para i en asc(1,n) haz escribe "+-----"; fin;
    escribe_linea "+";
    para j en asc(1,n) haz
      para i en asc(1,n) haz escribe "|",c[i,j]:4,' '; fin;
      escribe_linea "|";
      para i en asc(1,n) haz escribe "+-----"; fin;
      escribe_linea "+";
    fin para;
  fin imprime_resultados;

haz
  inicializa;
  repite
    pon_un_numero;
    siguiente_casilla;
    si esta_ocupada entonces rebota; fin;
  hastaque esta_lleno;
  imprime_resultados;
fin programa;
```

Algoritmo 23. Generacion de Cuadrados Magicos.

Generacion de una tabla de numeros primos, con una discusion sobre optimizacion de programas

En el Algoritmo 14 en la pagina 61 se estudia una condicion para determinar si un numero es o no primo. Una ampliacion de ese esquema, utilizando una

```
T: tabla [1..n] de entero;
```

para contener los primos (*T[i]* es el *i*-esimo primo), es la siguiente:

```
T[1] ← 1; T[2] ← 2; T[3] ← 3; -- los primeros primos
l ← 3; -- indice o lugar del ultimo primo calculado
mientras l < n haz
  si existe j en asc(T[l] + 2,infinito) talque
    no existe k en asc(2,l) talque j mod T[k] = 0 entonces
    l ← l + 1; T[l] ← j;
  fin si;
fin mientras;
```

Se utilizan dos existenciales anidados:

- el primero inspecciona cada numero a partir del ultimo primo mas 2 (mas 1 no tiene interes, ya que es un numero par; "infinito" es cualquier constante entera muy grande).
- y el segundo verifica si es primo probando si no es divisible exactamente por ninguno de los primos menores (que por construccion estan en la tabla *T*).

Una vez encontrado un numero primo, se mete en la tabla. Para ello, se dispone de una variable *l* entera, que por convencion siempre "marca" el indice del ultimo primo calculado; al encontrar uno nuevo, se incrementa ese indice, y se asigna a *T[l]* su valor, preservando asi la convencion sobre *l*. La iteracion se encarga de llenar la tabla de primos (hasta la constante *n*, tamaño de la tabla).

Un analisis detallado del programa muestra que se realizan bastantes calculos inutiles al verificar si un candidato a primo lo es o no, pues se halla su modulo respecto de todos los primos anteriores, mientras que solo es necesario hallarlo respecto de los primos que no excedan a su raiz cuadrada

> En efecto, si un candidato *j* es divisible exactamente por un primo *k* mayor que su raiz cuadrada, el cociente *j/k* es un divisor de *j* menor que su raiz cuadrada. *J/k* contendra algun factor primo (llamemosle *q*), tambien menor que *raiz(j)*; por todo lo cual, de ser *k* divisor de *j*, tambien lo es *q*, primo menor que *k*, que por construccion ya se ha analizado antes como divisor. De donde resulta que el calculo es superfluo.

Cambiaremos el segundo existencial por

```
no existe q en divisores talque j mod q = 0
```

definiendo la **secuencia**

```
secuencia divisores: entero haz
  k ← 2;
  repite
    produce T[k];
    k ← k + 1;
  hastaque T[k] * T[k] > j;
fin divisores;
```

Tambien puede observarse que *j* toma muchos valores que es superfluo verificar: en particular, todos los pares (para evitar el primer par se calcula precisamente *j* desde *T[l]+2* y no desde *T[l]+1*). Podrian tambien pensarse en evitar los multiplos de 3: bastaria con incrementar *p* alternativamente en 2 y 4 unidades, en vez de siempre 2; de todos modos, estos cambios son de detalle con respecto al realizado, en cuanto a la optimizacion del tiempo de proceso.

El programa final sera:

```
programa Numeros_primos es
  const n = 200; -- cantidad de numeros primos
  const infinito = 2000000000; -- mas o menos
  var p,q,j,k,l: entero;
  var T: tabla [1..n] de entero; -- los primos

  secuencia divisores: entero haz
    k ← 2;
    repite
      produce T[k];
      k ← k + 1;
    hastaque T[k] * T[k] > j;
  fin divisores;

haz
  T[1] ← 1; T[2] ← 2; T[3] ← 3; -- los primeros primos
  escribe_linea "Tabla de los primeros ",n," numeros primos.";
  escribe_linea 1:5;
  escribe_linea 2:5;
  escribe_linea 3:5;
  l ← 3; -- indice o lugar del ultimo primo calculado
  mientras l < n haz
    si existe j en asc(T[l]+2,infinito) talque
      no existe q en divisores talque j mod q = 0 entonces
      escribe_linea j:5;
      l ← l + 1; T[l] ← j;
    fin si;
  fin mientras;
fin programa;
```

Algoritmo 24. Generacion de numeros primos

El problema de la siguiente permutacion

Supuesto un conjunto C ordenado, p. ej. un conjunto de digitos

$C = \{0,1,2\}$

se llama *permutacion en C* a cualquier ordenacion de todos sus elementos.

La que sigue es una lista de todas las permutaciones posibles en C:

012 021 102 120 201 210

ordenadas en orden creciente.

El problema de la Siguiente Permutacion se plantea como sigue: dada una permutacion, hallar la siguiente en orden *alfabetico*, esto es, siguiendo el mismo criterio mediante el cual definimos una ordenacion entre las tiras de caracteres de igual longitud. Se notara que este criterio, aplicado a permutaciones numericas, equivale a la ordenacion numerica habitual; en el caso de caracteres, significa que ABC es menor que BAC, lo cual es logico. "La siguiente" significa la menor que supera a la dada.

Para determinar cual es la siguiente permutacion a una dada, realizaremos el siguiente analisis:

- En primer lugar, y supuesto el problema resuelto, es obvio que, entre una permutacion y la siguiente, habra una seccion inicial (posiblemente vacia) en que coincidan y otra en la que no. Nos fijamos en el primer elemento empezando por la izquierda en que no coinciden,

  034521 -- una permutacion
  035124 -- la siguiente
  '03' es invariable.
  El resto cambia.
  El primer elemento en que no coinciden esta en la posicion tercera: pasa de '4' a '5'.

y afirmamos que este elemento, principio de la parte derecha de la permutacion en que las dos diferiran, puede hallarse como el primer numero de la permutacion original empezando por la derecha y yendo hacia atras en el que la secuencia deja de ser creciente (en el ejemplo presentado, a partir de la primera permutacion, encontramos la serie '1', '2', '5', '4', y es '4' el primero que rompe la secuencia ascendente)

Debido a que, de estar situado mas a la derecha, la parte que deberia cambiar no podria hacerlo hacia una formacion mayor

Si intentamos cambiar a partir del 1, el 2 o el 5, es imposible encontrar una formacion mayor que la dada.

por estar esta ordenada decrecientemente, lo que le asigna la mejor posicion en la ordenacion; y de estarlo mas a la izquierda, forzosamente se realizaria un cambio mayor que el anunciado

No hay ninguna manera de, cambiando a partir del '0' o el '3', obtener una permutacion mayor que la primera y menor que la segunda.

- En segundo lugar, afirmamos que este primer elemento por la derecha que se altera debe intercambiarse por el menor numero de los que le siguen que le supera

  Ya que es la unica forma de obtener la menor permutacion

  y que el resto debe ordenarse en forma ascendente (que es justamente la minima entre las mayores).

En el ejemplo anterior:

034521

El primer elemento a cambiar es '4'
Debe intercambiarse por '5':

035421

Y el resto (421), ordenarse crecientemente

035124

lo cual equivale a darle la vuelta: 421 → 124.

El programa se plantea mediante una secuencia que produce permutaciones, de tipo

tipo *permutacion* **es aplicacion** *[1..10] ⇒ [0..9];*

donde una variable *n* indicara cuantos de los elementos de la permutacion se utilizan (p. ej., en el último ejemplo se utilizan 6 elementos).

El esquema de la secuencia es el siguiente:

```
produce p; -- permutacion inicial
mientras existe i en desc(n-1,1) talque P[i] < P[i+1]
  y entonces existe j en desc(n,i+1) talque P[j] > P[i] haz
  intercambia_i_y_j;
  ordena_resto;
  produce p;
fin mientras;
```

Los existenciales resumen el proceso de identificacion del lugar de los digitos que deben intercambarse. El resto es trivial en diseño descendente.

```
programa permutaciones es

  var n: entero;  -- permutaciones de los 'n' primeros numeros desde 0

  tipo permutacion es aplicacion [1..10] => [0..9];  -- 1 ≤ n ≤ 10

  var P: permutacion;

  secuencia permutaciones: permutacion es
    var i,j: entero;

    accion intercambia_i_y_j es var aux: entero; haz
      aux ← P[i]; P[i] ← P[j]; P[j] ← aux;
    fin intercambia_i_y_j;

    accion ordena_resto haz
      i ← i + 1; j ← n;
      mientras i < j haz intercambia_i_y_j; i ← i + 1; j ← j - 1; fin;
    fin ordena_resto;

  haz
    produce p; -- permutacion inicial
    mientras existe i en desc(n-1,1) talque P[i] < P[i+1]
      y entonces existe j en desc(n,i+1) talque P[j] > P[i] haz
      intercambia_i_y_j;
      ordena_resto;
      produce p;
    fin mientras;
  fin permutaciones;

  accion escribe_permutacion es var i: entero; haz
    para i en asc(1,n) haz escribe P[i]; fin;
    escribe_linea;
  fin escribe_permutacion;

  accion inicializa es var i: entero; haz
    itera
      escribe_linea "De cuantos elementos quieres realizar permutaciones?"; lee n;
    sal cuando n > 1 y n < 11;
      escribe_linea "No vale, ha de estar entre 2 y 10.";
    fin itera;
    para i en asc(1,n) haz P[i] ← i - 1; fin;
  fin inicializa;

haz

  inicializa;
  para p en permutaciones haz escribe_permutacion; fin;

fin programa;
```

Algoritmo 25. Generacion de permutaciones de N elementos por el metodo de la Siguiente Permutacion

Manipulacion de Matrices

El siguiente programa define acciones y funciones para la manipulacion de matrices. Todas ellas son elementales, pero constituyen un vocabulario suficientemente abstracto para escribir cualquier tipo de algoritmo de manejo de matrices. El programa en si se limita a demostrar la utilizacion de esos subprogramas. Se utilizan parametros por constante para evitar copias innecesarias de datos.

```
programa manejo_de_matrices es

  const dim = 3; -- dimension de las matrices
  tipo indice es [1..dim];
  tipo matriz es tabla indice,indice de real;

  funcion suma (const a,b: matriz): matriz es
    var m: matriz; i,j: indice;
  haz
    para i en asc(1,dim) haz para j en asc(1,dim) haz m[i,j] ← a[i,j] + b[i,j]; fin; fin;
    vale m;
  fin suma;

  funcion resta (const a,b: matriz): matriz es
    var m: matriz; i,j: indice;
  haz
    para i en asc(1,dim) haz para j en asc(1,dim) haz m[i,j] ← a[i,j] - b[i,j]: fin; fin;
    vale m;
  fin resta;

  funcion mult (const a,b: matriz): matriz es
    var m: matriz; i,j,k: indice;
  haz
    para i en asc(1,dim) haz para j en asc(1,dim) haz
      m[i,j] ← 0;
      para k en asc(1,dim) haz m[i,j] ← m[i,j] + a[i,k] * b[k,j]; fin;
    fin para; fin para;
    vale m;
  fin mult;

  accion lee_matriz (var m: matriz) es
    var i,j: indice;
  haz
    para i en asc(1,dim) haz para j en asc(1,dim) haz lee m[i,j]; fin; fin;
  fin lee_matriz;

  accion escribe_matriz (const m: matriz) es
    var i,j: indice;
  haz
    para i en asc(1,dim) haz
      para j en asc(1,dim) haz escribe ' ',m[i,j]; fin; escribe_linea;
    fin para;
  fin escribe_matriz;

  var A,B: matriz;

haz

  escribe_linea "Escribe ",dim*dim,
    " numeros reales para dar valor a la primera matriz:";
  lee_matriz A;

  escribe_linea "Escribe ",dim*dim,
    " numeros reales para dar valor a la segunda matriz:";
  lee_matriz B;

  escribe_linea "Valor de la suma:"; escribe_matriz suma(A,B);

  escribe_linea "Valor de la resta:"; escribe_matriz resta(A,B);

  escribe_linea "Valor del producto:"; escribe_matriz mult(A,B);

fin programa;
```

Algoritmo 26. Manejo de matrices

El problema de las ocho reinas

Se dispone de ocho reinas del juego de Ajedrez, y se trata de ponerlas en el tablero de modo que no se maten entre si. Interesa hallar todas las formas posibles de hacerlo.

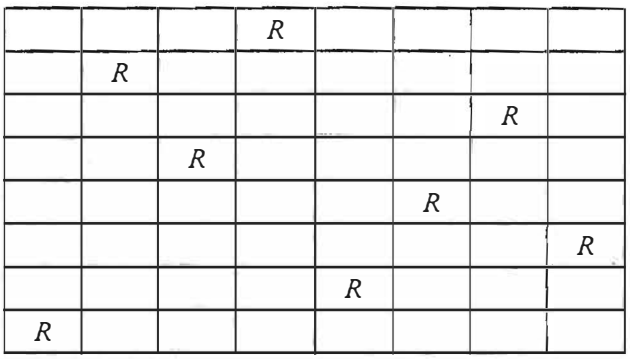


Figura 63. Una de las soluciones al problema de las Ocho Reinas

Una primera aproximacion a la resolucion del problema podria consistir en probar todas las disposiciones posibles de ocho reinas en el tablero, eliminando las imposibles (como "todas las reinas en la casilla inferior izquierda") y las erroneas (aquellas en las que se matan entre si).

```
para reina1 en posiciones_del_tablero haz
  para reina2 en posiciones_del_tablero haz
    ...
      para reina8 en posiciones_del_tablero haz
        si no se_matan entonces solucion;
        sino ...
        fin: si;
      fin para;
    ...
  fin para;
fin para;
```

Si estimamos el numero de combinaciones que habria que verificar, obtenemos

Cada reina puede colocarse en 64 casillas distintas
Hay 8 reinas
Total = 64^8 = 281474976710656 combinaciones

que, evidentemente, excede la capacidad de cualquier interprete.

Se obtiene una reduccion notable del numero de combinaciones si se considera que, dado que dos reinas que se hallen en la misma fila se matan entre si, no se pierden soluciones si convenimos en asignar una reina a cada una de las filas, permitiendole moverse solo dentro de esa fila.

```
para reina1 en fila1 haz
  para reina2 en fila2 haz
    ...
      para reina8 en fila8 haz
        si no se_matan entonces solucion;
        sino ...
        fin: si;
      fin para;
    ...
  fin para;
fin para;
```

El numero de combinaciones en este caso es

Cada reina puede colocarse en 8 casillas distintas
Hay 8 reinas
Total = 8^8 = 16777216 combinaciones

y se considera tambien excesivo, aunque ya es manejable por un ordenador grande.

Un analisis mas fino del problema permite ver que una vez colocadas algunas de las reinas, no tiene sentido probar de colocar las demas en lugares en los que se maten con las anteriores; y sugiere un programa que vaya "probando" las distintas posibilidades de colocacion, y deseche inmediatamente las que provoquen conflictos (el metodo es mas general, se aplica como tecnica de construccion de algoritmos, y se conoce como *backtracking* o *retroceso*).

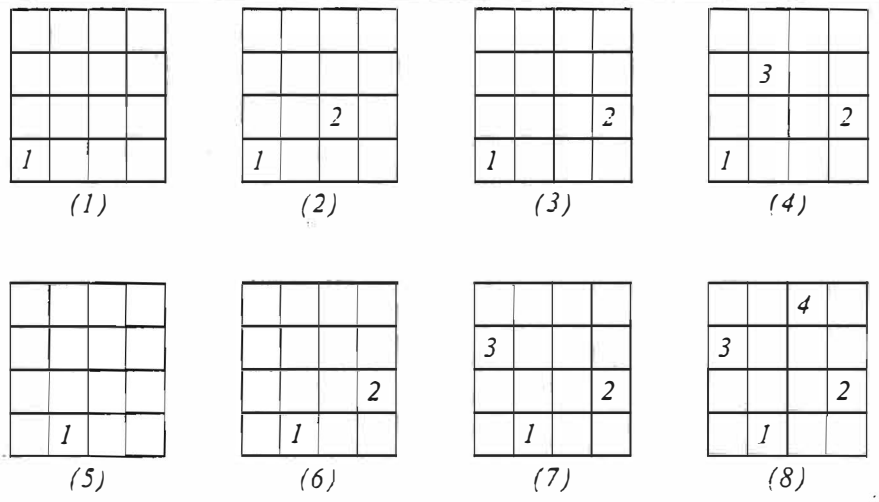


Figura 64. ***Fragmento del proceso de Backtracking para las Ocho Reinas en un tablero de 4x4:*** (1) representa el primer paso del proceso: la reina 1 se pone en la primera casilla de la fila 1. En (2), se ha colocado la reina 2 en la primera casilla de la fila 2 en la que no se mata con otra (ya que probar aquellas en las que se mata no tiene sentido). Se observa que no es posible colocar ninguna reina que no se mate en la fila 3, de lo que se deduce que la posicion de la reina 2 es equivocada, cambiandola a la de (3) [correccion]. En (4) se coloca la reina 3 en la unica posicion posible: resulta que ello no permite poner ninguna reina en la fila 4; como no es posible poner en otro lugar la reina 3, se elimina [retroceso], y tambien la 2, que ya no puede ponerse en otro sitio. La posicion de la reina 1 es pues incorrecta, y se prueba con la mostrada en (5). (6), (7) y (8) colocan las reinas en las primeras posiciones libres disponibles, obteniendo esta vez una solucion al problema (8).

El programa esta basado en el algoritmo descrito. En cada paso se controla que columnas y diagonales estan "ocupadas", en el sentido de que no es posible poner ahi una reina porque se mataria con una ya puesta. La numeracion de las columnas es la usual, de 1 a 8; las diagonales se numeran basandose en que la suma de la fila y la columna de las casillas situadas en la misma diagonal descendente es constante, como lo es la resta para las diagonales ascendentes. Las variables

```
var da: aplicacion [-7..7] => logico;
var dd: aplicacion [2..16] => logico;
var col: aplicacion [1..8] => logico;
```

permiten el control descrito.

Del tablero solo nos interesa la posicion de cada reina en cada fila, por lo que se representa como

```
var D: tabla [1..8] de [1..8];
```

La fila en la que se intenta poner la reina es

```
var f: [0..8];
```

y toma el valor 0 al final del proceso (esto es, cuando, halladas ya todas las soluciones, se decide que la reina numero 1 debe eliminarse).

El resto del programa esta tomado directamente de la descripcion de la Figura 64.

```
programa Ocho_reinas es

 var f:[0..8]; acabo_de_poner_reina: logico;
 var da: aplicacion [-7..7] => logico; dd: aplicacion [2..16] => logico;
 var col: aplicacion [1..8] => logico; D: tabla [1..8] de [1..8]: candidato: [1..9];

 accion pon_la_primera_reina es var i: entero; haz
  f ← 1; D[1] ← 1; acabo_de_poner_reina ← cierto;
  para i en asc(-7,7) haz da[i] ← falso; fin; da[0] ← cierto;
  para i en asc(2,16) haz dd[i] ← falso; fin; dd[2] ← cierto;
  para i en asc(1,8) haz col[i] ← falso; fin; col[1] ← cierto;
 fin pon_la_primera_reina;

 condicion hay_8 haz vale f = 8; fin; condicion quedan_reinas haz vale f > 0; fin;

 accion solucion es var i: entero; haz
  para i en asc(1,8) haz escribe ' ',D[i]; fin; escribe_linea;
 fin solucion;

 accion saca_reina haz
  si candidato < 9 entonces da[f-D[f]] ← falso; dd[f+D[f]] ← falso; col[D[f]] ← falso; fin;
  f ← f - 1; acabo_de_poner_reina ← falso;
 fin saca_reina;

 condicion puedo_poner_otra haz
  candidato← 1; f ← f + 1;
  repite
   si no col[candidato] y no da[f-candidato] y no dd[f+candidato] entonces vale cierto; fin;
   candidato ← candidato + 1;
  hastaque candidato = 9;
  f ← f - 1; vale falso;
 fin puedo_poner_otra;

 accion pon_reina haz
  col[candidato] ← cierto; da[f-candidato] ← cierto;
  dd[f+candidato] ← cierto; D[f] ← candidato;
 fin pon_reina;

 condicion puedo_moverla haz
  da[f-D[f]] ← falso; dd[f+D[f]] ← falso; col[D[f]] ← falso; candidato ← D[f] + 1;
  mientras candidato < 9 haz
   si no col[candidato] y no da[f-candidato] y no dd[f+candidato] entonces
    acabo_de_poner_reina ← cierto; vale cierto;
   fin si;
   candidato ← candidato + 1;
  fin mientras;
  vale falso;
 fin puedo_moverla;

haz
 pon_la_primera_reina;
 repite
  si acabo_de_poner_reina entonces
   decide
    cuando hay_8 => solucion; saca_reina;
    cuando puedo_poner_otra => pon_reina;
    cuando otros => acabo_de_poner_reina ← falso;
   fin decide;
  sino si puedo_moverla entonces pon_reina; sino saca_reina; fin;
  fin si;
 hastaque no quedan_reinas;
fin programa;
```

Algoritmo 27. Las Ocho Reinas

Algoritmos de ordenacion

Sea *I* un tipo ordenado apto para ser indice de una **tabla** o origen de una **aplicacion**. con limites *min* y *max*:

tipo *I* **es** *[min..max];*

y *O* un tipo provisto de todas las operaciones de comparacion. Se dice que un objeto

tipo *vector* **es tabla** *I* **de** *O;*
var *T: vector;*

esta *ordenado* si, al recorrer la tabla, los elementos *T[j]* se encuentran en orden creciente; o, mas formalmente,

{1} T esta ordenado ⇔ **para** *j,k* **en** *I, j* < *k* ⇒ *T[j]* < *T[k]*

Frecuentemente se plantea la necesidad de ordenar una tabla, es decir, de intercambiar o reordenar algunos de sus elementos hasta conseguir una tabla ordenada. Presentamos a continuacion algunos algoritmos de ordenacion.

Aproximacion inicial: La formula {1} puede ser expresada con mas rigor como

{1b} T esta ordenado ⇔ **para** *j* **en** *I* **para** *k* **en** *I* **talque** *k* > *j: T[j]* < *T[k]*

y de esta formulacion se deriva directamente el siguiente programa: simplemente, se "arregla" cada par que haga {1b} incorrecta.

```
accion ordena(var T: vector) es
  var aux: O; j,k: I;
haz
  para j en asc(min,pred(max)) haz
    para k en asc(suc(j),max) talque T[j] > T[k] haz
      aux ← T[j]; T[j] ← T[k]; T[k] ← aux;
    fin para;
  fin para;
fin ordena;
```

Algoritmo 28. Ordenacion de vectores, metodo elemental

De otro modo: se compara el primer elemento con todos los que le siguen, intercambiando cada par incorrectamente ordenado. Con ello se consigue, como minimo, llevar el menor elemento al principio del vector; a continuacion, se repite el proceso con los demas elementos.

Metodo de la Burbuja: Consiste en imaginar el vector colocado en vertical, y una "burbuja ordenadora" que sube varias veces a lo largo del vector. La burbuja abarca dos elementos contiguos, y los intercambia si estan mal ordenados.

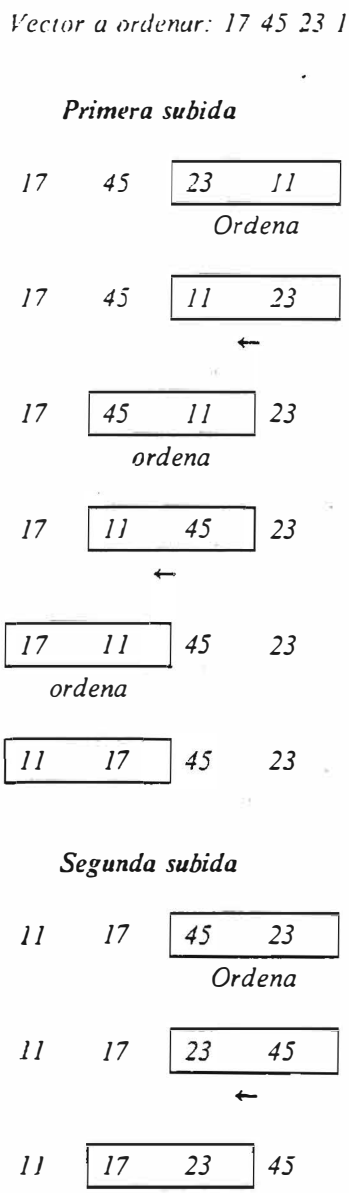


Figura 65. ***Ordenacion por el metodo de la burbuja:*** La burbuja se representa mediante una cajita; se supone que "arriba" es a la izquierda, y "abajo" la derecha. En este caso solo se necesita hacerla subir dos veces. Cada vez que pasa, ordena si es necesario.

Cada vez que la burbuja sube, solo necesita hacerlo hasta un lugar menos que la vez anterior, ya que el menor elemento es llevado con seguridad a la primera posicion (la mas alta) por la burbuja en cada vuelta.

El algoritmo resulta directamente de la explicacion dada

```
accion ordena( var T: vector ) es
  var j,k: I; aux: O;
haz
  para j en asc(suc(min),max) haz -- J: subidas
    para k en desc(max,j) talque T[k-1] > T[k] haz -- K: lugar de la burbuja
      aux ← T[k-1]; T[k-1] ← T[k]; T[k] ← aux;
    fin para;
  fin para;
fin ordena;
```

Algoritmo 29. Ordenacion de vectores por el metodo de la burbuja

Busqueda de elementos en una tabla

Otro problema que tambien se presenta con frecuencia es el de la *busqueda* de un elemento que tome determinado valor x (con mas propiedad, lo que se busca es su subindice). En el caso de un vector no ordenado no hay mas remedio que realizar una inspeccion exhaustiva:

```
si existe i en asc(min,max) talque T[i] = x entonces
  el_subindice_es I;
sino no_hay_un_tal_subindice;
fin si;
```

Si el vector esta ordenado, un metodo mucho mas rapido es el de *busqueda binaria*, basado en la misma idea que el metodo de biparticion presentado en el Algoritmo 13 en la pagina 57. Suponemos que el tipo indice *I* es numerico para simplificar (sino, se podria recurrir a conversiones de tipo), y presentamos el algoritmo en forma de **accion** con dos parametros: el segundo, de tipo logico, indica si existe el valor en la tabla o no, y el primero contiene su indice si el segundo vale *cierto*.

```
accion busqueda_binaria(var indice: I; var encontrado: logico) es
  -- busca X en la tabla T
  var j,k,m: I;
haz
  j ← min; k ← max;
  repite
    m ← (j+k) div 2; -- punto medio
    si X < T[m] entonces j ← m - 1; sino k ← m + 1; fin;
  hastaque T[m] = X o j > k;
  indice ← m;
  encontrado ← no (j > k);
fin busqueda_binaria;
```

Algoritmo 30. Busqueda binaria

Ejercicios

1. Dada una lista de numeros terminada por un cero, escribirla ordenada, primero en orden creciente, y luego en orden decreciente.

2. Escribir un conjunto de subprogramas analogos a los del Algoritmo 26 en la pagina 99 para tratar vectores en el espacio ordinario, incluyendo suma, resta, producto escalar, producto por un escalar y norma.

3. Hacer un programa que evalue polinomios: los coeficientes se introducen como serie terminada por un cero; a continuacion van valores de x, tambien en forma de serie con cero, para cada uno de los cuales se evalua el polinomio. Una serie vacia termina la ejecucion del programa.

4. Se dispone de la siguiente tabla:

1	I	11	XI	21	XXI	100	C	1000	M		
2	II	12	XII	22	XXII	200	CC	2000	MM		
3	III	13	XIII	28	XXVIII	300	CCC	10000	x		
4	IV	14	XIV	29	XXIX	400	CD	100000	c		
5	V	15	XV	30	XXX	500	D				
6	VI	16	XVI	40	XL	600	DC				
7	VII	17	XVII	50	L	700	DCC				
8	VIII	18	XVIII	60	LX	800	DCCC				
9	IX	19	XIX	70	LXX	900	CM				
10	X	20	XX	80	LXXX						
90	XC										

a la que su autor no añade ninguna informacion suplementaria. Las minusculas indican mayusculas "superralladas". Escribir un programa que convierta un numero entero a su representacion romana, y viceversa.

5. Dado un texto en el que intervienen no mas de 100 palabras distintas, escribir una tabla de frecuencias de aparicion de esas palabras.

6. Extender el Algoritmo 27 en la pagina 102 para que solo imprima las soluciones esencialmente distintas, es decir, las que no sean giros, simetrias o inversiones una respecto de otra.

7. Dado un numero entero, descomponerlo en sus factores primos.

Tuplas

Productos Cartesianos

El *producto cartesiano* de dos conjuntos A y B es el conjunto de los pares ordenados (a,b), con a **en** A y b **en** B:

$$A \times B = \{(a,b) \mid a \textbf{ en } A \textbf{ y } b \textbf{ en } B\}$$

Dado un par (a,b), se llama *proyeccion sobre* A o *primera proyeccion* al elemento a, y proyeccion sobre B o segunda proyeccion al elemento b; A y B son las *componentes* del par.

El concepto de producto cartesiano se extiende facilmente para productos de varios conjuntos, e igualmente sucede con las proyecciones.

Uniones Disjuntas

Dados dos conjuntos A y B, se llama *union disjunta* de A y B al conjunto

$$A \oplus B = \{(a,\{A\}),\ a \textbf{ en } A\} \cup \{(b,\{B\}),\ b \textbf{ en } B\} = (A \times \{A\}) \cup (B \times \{B\})$$

Alternativamente: la union disjunta de A y B contiene los elementos de A distinguidos con el subindice "A" y los de B con el subindice "B".

Para conjuntos A y B disjuntos, el resultado es equivalente a la union de A y B, pero para conjuntos no disjuntos, el subindice permite diferenciar el conjunto de origen de cada elemento:

La union de los naturales (N) y los reales (R) es R, pero su union disjunta es

$$\{(a,b) \mid (a \textbf{ en } N \textbf{ y } b = \{N\}) \textbf{ o } (a \textbf{ en } R \textbf{ y } b = \{R\})\}$$

Esta definicion se generaliza inmediatamente a uniones disjuntas de varios conjuntos.

Creacion de un tipo para representar numeros complejos

A partir del tipo predefinido *real*, deseamos construir un tipo *complejo* cuyo conjunto de valores permita representar los numeros complejos. Dado que el conjunto C de los numeros complejos puede estudiarse como producto cartesiano del conjunto de los reales (R) por si mismo, $R \times R$, estamos interesados en construir un tipo cuyos valores sean *pares ordenados* o *2-tuplas* de reales. Los matematicos hablan, dado un valor de $R \times R$ como $(3,5)$, de la *primera componente* (3) y la *segunda componente* (5) de la tupla; en UBL, denominaremos *campos* a los componentes, y utilizaremos identificadores en vez de numeros para distinguirlos: en este ejemplo, llamaremos *re* a la parte real (primera componente) e *im* a la parte imaginaria (segunda componente) del numero complejo.

Definiremos el tipo *complejo* como sigue:

tipo *complejo* **es tupla** *re:real; im:real;* **fin tupla**;

o, abreviadamente,

tipo *complejo* **es tupla** *re,im: real;* **fin**;

A partir de esta definicion, podremos declarar objetos de tipo complejo

var *c,c1,c2: complejo;*

y realizar operaciones de asignacion o comparacion de (des)igualdad entre ellos:

c ← c1;

si *c2 ≠ c1* **entonces** ...

No es posible comparar tuplas mediante los operadores $<$, $\leq$, $>$, $\geq$, ya que no existe una ordenacion natural o canonica de un producto cartesiano (sin embargo, podemos definir nuestras propias funciones y condiciones que realicen algunas de estas operaciones, si lo consideramos adecuado):

condicion *menor(c1,c2: complejo)* **haz** -- *en valor absoluto*
  vale *c1.re*c1.re + c1.im*c1.im < c2.re*c2.re + c2.im*c2.im;*
fin *menor;*

Para acceder a los campos *re* e *im* del complejo *c* (primera y segunda componente) utilizaremos la notacion

c.re y *c.im*

La operacion de seleccionar un campo entre los que componen un objeto de tipo tupla es un modo de *seleccion*. En nuestro caso, estos campos tienen tipo *real*, y pueden ser utilizados en cualquier ocasion en la que se permita una *variable* real; en particular, es posible realizar asignaciones, comparaciones (sin restriccion), operaciones aritmeticas, etc.

para asignar el complejo *(2,3)* al objeto *c*, utilizaremos

c.re ← 2; c.im ← 3;

Para poder trabajar con la nocion de suma de numeros complejos, definiremos la siguiente **funcion**:

---

funcion *suma(a,b: complejo): complejo* **es**
  var *resultado: complejo;*
haz
  resultado.re ← a.re + b.re; -- *"la parte real es la suma de las partes reales...*
  resultado.im ← a.im + b.im; -- *... y la parte imaginaria, la suma de las partes imaginarias."*
  vale *resultado;*
fin *suma;*

Algoritmo 31. Suma de numeros complejos

Estructura de los objetos de tipo tupla

Los valores que toman los objetos de los diferentes tipos pueden ser *simples* o *estructurados*. Se han estudiado ya varias formas de tipos y valores estructurados: tiras, conjuntos y tablas son formas de obtener estructuras de datos *homogeneas*, en el sentido de que todas las componentes o subpartes de sus valores son del mismo tipo; en el caso de las **tuplas**, pueden crearse tipos cuyos valores sean *heterogeneos*, ya que cada campo puede tener su propio tipo, posiblemente distinto de los demas tipos que intervienen en la **tupla**.

```
tipo dia es [1..31];

tipo mes es
  (enero,febrero,marzo,abril,mayo,junio,julio,agosto,
  septiembre,octubre,noviembre,diciembre) ciclico;

tipo año es [0..2000];

tipo fecha es tupla d: dia; m: mes; a: año; fin;

var f1, f2: fecha;

-- Los siguientes pueden ser valores de F1 y F2 durante la ejecucion del programa

(*
  f1
| 21 | Diciembre | 1978 |
  f1.d    f1.m      f1.a

  f2
| 3 | Enero | 1984 |
  f2.d    f2.m    f2.a
*)

accion lee_fecha(var f: fecha) haz
  lee f.d,f.m,f.a;
fin lee_fecha;

accion escribe_fecha(const f: fecha) haz
  escribe f.d,' de ',f.m,' de ',f.a;
fin escribe_fecha;

condicion menor(const f1,f2: fecha) haz
  vale f1.a < f2.a o sino
    (f1.a = f2.a y entonces
      (f1.m < f2.m o sino
        (f1.m = f2.m y entonces f1.d < f2.d)));
fin menor;
```

Figura 66. Declaraciones y operaciones sobre un tipo fecha

Tipo tupla. Nombres de campo

```
tipo_tupla = tupla_fija | tupla_con_variantes

tupla_fija =
  tupla
    campo {,campo} : tipo;
    {campo {,campo} : tipo;}
  fin [tupla]

campo = identificador

Estilo: ademas del sugerido en la sintaxis,

tupla campo(s): tipo; {campo(s): tipo} fin;
```

Figura 67. Sintaxis y estilo de un tipo tupla

Los identificadores de los campos pueden ser identicos a otros declarados en el programa, o predefinidos, aunque han de ser distintos entre si; dado que (excepto con el uso de la instruccion **con**, que se estudiara mas adelante en este mismo capitulo) el unico modo de referirse a un campo es mediante la operacion de seleccion, no hay ambiguedad posible en la identificacion.

```
var x: real;
var z: tupla x: entero; fin;
```

No hay peligro de ambiguedad, ya que *x* siempre se refiere a la variable real *x*, y el campo *x* de *z* debe escribirse *z.x*.

Las operaciones predefinidas aplicables a los objetos de tipo **tupla** son la asignacion, la comparacion por (des)igualdad, y la *seleccion*:

variable_tupla.campo

se refiere al campo con nombre *campo* del objeto *variable_tupla*. Los campos actuan como objetos variables, y pueden ser utilizados en cualquier contexto en el que se requiera un objeto variable de su tipo. Son ejemplos de campo

```
x.a      -- campo a de la tupla x
y.b.c    -- campo c de la tupla b, que a su vez es campo de la tupla y
x[3].c   -- campo c de la tupla x[3]. Por lo visto x es una tabla de tuplas
```

El lenguaje permite declarar campos de tipo tupla (o, en general, de cualquier otro tipo), asi como se permiten tablas de tablas, tuplas de tablas, o cualquier combinacion:

```
tipo T es tabla [1..100] de
  tupla
    a: tabla [1..6] de tira(20);
    b: conjunto de [10..19];
    c: tupla
         d: entero;
         e: real;
         f: (g,h,i,j,k) ciclico;
       fin tupla;
  fin tupla;
```

Puede asignarse un valor a un objeto de tipo tupla asignando valores individualmente a cada uno de los campos:

```
tipo digrafo es tupla c1,c2: caracter; fin;
-- define un tipo digrafo cuyos valores son pares de caracteres

var d: digrafo; -- declara un objeto variable de tipo digrafo

d.c1 ← ' '; d.c2 ← ' '; -- asigna el par (' ',' ') a d
```

Un objeto de tipo tupla no tiene valor hasta que lo tienen cada uno de sus campos.

Tuplas con Variantes

Las **tuplas** se utilizan a menudo para diseñar tipos cuyos valores agrupan informacion sobre objetos, al modo de las fichas de un fichero de personal o de los datos de un carnet de identidad. En muchos casos sucede que parte de la informacion es opcional (en una ficha: solo debe ser llenada si se cumplen algunos requisitos en los campos anteriores [pues de otro modo no tendria sentido]; en UBL: algunos campos deben existir solo si otros campos tienen determinado valor). Las **tuplas** con variantes permiten diseñar tipos de datos apropiados a esas estructuras.

```
tupla_con_variantes =
  tupla
    {campo {,campo}: tipo;}
    parte_variante
  fin [tupla];

parte_variante =
  segun discriminante: tipo_del_discriminante es
    cuando valor_o_rango { "|" valor_o_rango} =>
      {campo {,campo}: tipo;}
      [parte_variante]
    {cuando valor_o_rango { "|" valor_o_rango} =>
      {campo {,campo}: tipo;}}
      [parte_variante]
    [cuando otros =>
      {campo {,campo}: tipo;}]
      [parte_variante]
  fin [segun];

valor_o_rango = expresion_constante [ .. expresion_constante]

discriminante = identificador

tipo_del_discriminante = identificador
```

Figura 68. ***Sintaxis de una*** **tupla** ***con variantes:*** Notese que la sintaxis de una *parte_variante* especifica que esta puede contener a su vez otras *parte_variantes*

Llamaremos *campos fijos* a los que aparecen antes de **segun**, y *campos variantes* a los que aparecen despues. Los campos variantes existiran segun el valor del *campo discriminante* (o, simplemente, *discriminante*).

Accesibilidad y existencia de los campos variantes

Una **tupla** con variantes se utiliza como una **tupla** normal, con la particularidad de que cada uno de los campos variantes solo existe (y, por tanto, solo es accesible) cuando el valor del discriminante (que, por lo demas, es un campo como cualquier otro) coincide con el valor de la correspondiente expresion constante:

> Deseamos construir un tipo que permita centralizar la informacion que poseemos sobre determinadas personas. De cada persona conocemos su nombre, la fecha de su nacimiento, su estado civil, y, solo en el caso de que este casada, el nombre de su pareja. Definimos el **tipo** *persona* como:

```
tipo estado_civil = {soltero,casado};

tipo nombre_completo es tira(20);

tipo persona es
  tupla
    n: nombre_completo;
    nace: fecha;
    segun estado: estado_civil es
      cuando casado => pareja: nombre_completo;
      cuando soltero => ; -- no hay nombre si no esta casado
    fin segun;
  fin tupla;

var p1,p2: persona;
```

p1

EMMA SANCHEZ RIUS		
21	*Enero*	*1950*
Soltero		

p2

ANA GOMEZ GOMEZ		
7	*Marzo*	*1957*
Casado		
PEDRO LOPEZ CUETO		

Figura 69. ***Declaraciones y graficos para un tipo*** **tupla** ***con variantes:*** Los graficos muestran posibles valores de *p1* y *p2*. El campo *pareja* solo existe en el caso de *p2*, debido a que el valor del discriminante es *Casado*.

Las tuplas con variantes como uniones disjuntas

Asi como el concepto de producto cartesiano tiene su correspondencia en el lenguaje UBL en el concepto de **tupla**, el concepto de union disjunta puede representarse mediante tuplas con variantes sin campos fijos:

Para crear un tipo cuyos valores sean o reales o enteros (esto es, cuyo conjunto de valores sea la union disjunta de los reales y los enteros), definiremos

```
tipo entero_o_real es
  tupla segun es_real: logico es
    cuando cierto => r: real;
    cuando falso => e: entero;
  fin segun; fin tupla;
```

Como todos los objetos, las tuplas con variantes (y en particular, sus campos y su discriminante) no tienen valor antes de asignarles alguno. De este modo, es un error acceder a un campo variante sin que el discriminante tenga el valor adecuado:

```
x: entero_o_real;
...
x.e ← 5; -- error si no se verifica x.es_real = falso
```

Los campos variantes correspondientes a determinado valor del discriminante dejan de existir al cambiar este valor: no puede suponerse que el valor de un campo variante se conservara entre cambios del discriminante:

```
x: entero_o_real;
...
x.es_real ← cierto; x.r ← 1.0;
x.es_real ← falso; x.e ← 3;
x.es_real ← cierto;
-- despues de la ejecucion de esta instruccion no tiene porque verificarse que x.r = 1.0
```

Instruccion con

La instruccion **con** sirve para poder referirse directamente a los campos de un objeto dado de tipo tupla, sin necesidad de indicar directamente en cada instruccion el nombre del objeto.

---

instruccion_con =
con *variable_tupla* {,*variable_tupla*} **haz**
instruccion
{*instruccion*}
fin [**con**];

variable_tupla = *variable*

Estilo: ademas del sugerido en la sintaxis,

con *var* {,*var*} **haz** *instr* {*instr*} **fin**;

***Figura 70. Sintaxis y estilo de la instruccion* con**

La instruccion **con** calcula el nombre del objeto tupla (que puede ser subindiciado o campo de otra tupla, por ejemplo), y ejecuta las instrucciones subordinadas (que constituyen lo que llamaremos su *ambito*), aplicando para la identificacion de los objetos las siguientes convenciones:

- En el ambito de la instruccion **con**, los campos de la *variable_tupla* pueden utilizarse libremente como si fueran variables, sin necesidad de prefijarlos con el nombre de la tupla.

  var *f: fecha;*
  ...
  con *f* **haz** *d* ← *28; m* ← *Febrero; a* ← *1999;* **fin**;
  – *es una forma rapida y comoda de dar valor a* f. *Equivale a*
  f.d ← *28; f.m* ← *Febrero; f.a* ← *1999;*

- Si se produce alguna ambiguedad entre un campo y otra entidad en el ambito de la instruccion **con**, se resuelve esta en favor de la *variable_tupla*:

  var *f: fecha; d: caracter;*
  ...
  con *f* **haz**
  – *cualquier referencia a* d *accede a* f.d *y no al caracter* d *en este ambito*
  fin con;
  – *Aqui* d *es el caracter y no* f.d

  En este sentido, la instruccion **con** puede *esconder* algunos identificadores.

- Se toma

  con O_1, O_2 **haz** *S* **fin con**;

  como equivalente a

  con O_1 **haz con** O_2 **haz** *S* **fin con**; **fin con**;

  y, en este caso, como en aquel en el que entre **haz** y **con** hay mas instrucciones, si los tipos de O_i y O_j coinciden, la ambiguedad resultante de intentar decidir si un campo *c* es de O_i o de O_j se resuelve en favor del *O* mas cercano textualmente entre los que preceden a la instruccion en cuestion; de otro modo, se toma el O_n con mayor *n*.

```
var f1,f2: fecha.
  ...
con f1 haz
  instruccion_1;
  con f2 haz
    -- A es F2.A y no F1.A
  fin con;
  -- A es F1.A
fin con;
```

En los siguientes capitulos se encontraran mas ejemplos de programacion con **tuplas**.

Filas

Las **filas** son estructuras de datos que permiten acceder desde un programa a *ficheros de datos* contenidos en *dispositivos de almacenamiento externo* (llamados tambien a veces *perifericos*), como cintas magneticas, discos, fichas perforadas, papel impreso u otros medios. Las posibilidades ofrecidas por los distintos perifericos varian segun la naturaleza de estos (por ejemplo, normalmente puede modificarse desde programa un fichero en disco magnetico, pero no uno fichas perforadas); aqui nos limitaremos a una presentacion de lo que llamaremos *filas secuenciales*.

Un fichero puede concebirse como una sucesion de objetos, de longitud variable.

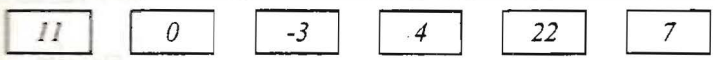


Figura 71. *Un fichero de enteros*

Estos objetos se encuentran fisicamente "fuera" del alcance del interprete del programa, excepto mediante las **filas**, metodos de acceso a los ficheros, y sus operaciones, que se definiran; su localizacion o *ubicacion* podra variar entre dos ejecuciones de un programa, sin afectar a sus resultados si los datos son los mismos, y podra ser definida exteriormente al programa o mediante la accion predefinida *conecta*.

Acceso a los componentes de un fichero; estructura de fila

Distinguiremos los *ficheros*, grupos de datos contenidos en algun medio externo de almacenamiento, de las **filas**, estructuras del lenguaje que permitiran el acceso a los valores de un fichero.

Las estructuras de datos estudiadas hasta el momento permiten el *acceso aleatorio* a sus componentes, en el sentido de que (salvo restricciones asociadas al valor del discriminante en el caso de las **tuplas** con variantes) es posible referirse a cualquier campo o elemento de una **tupla**, **tabla**, **aplicacion** o **conjunto** sin limitacion alguna en el orden de acceso (por ejemplo, despues de acceder a *T[1]* puede accederse a *T[i]*, para cualquier *i* valido). En cambio, las **filas** son objetos que proporcionan un *acceso secuencial* y controlado a los elementos de un fichero.

El fichero de la Figura 71 puede manipularse mediante una variable de tipo **fila**:

var *f:* **fila de** *entero;*

Convendremos en que, en cada momento de la ejecucion del programa, solo es accesible a traves de la **fila** un unico elemento del fichero, que llamaremos *ventana* o "*buffer*" de la **fila**; de otro modo, es como si "mirasemos" el fichero mediante una ventana, que forma parte de la fila. En la ventana solo cabe un elemento del fichero.

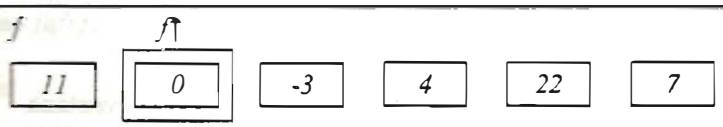


Figura 72. *El mismo fichero, su* **fila** *y la ventana:* (que en este caso esta situada "sobre" el segundo elemento).

Para referirnos a la ventana, utilizaremos la notacion *f*↑(si *f* es la **fila** en cuestion). *F*↑ es un modo de *seleccion*. y por tanto puede utilizarse como cualquier objeto de su tipo: *T* si *f* es **fila de** *T*.

tipo_fila = **fila de** *tipo*

Figura 73. Sintaxis de un tipo fila

El efecto combinado de la inspeccion o modificacion de la ventana y las acciones predefinidas que se explicaran, que "corren a la derecha" la ventana, permitiran la inspeccion (o *lectura*) o la creacion (o *escritura*) de cualquier fichero.

Distinguiremos dos tipos o *modos* de tratamiento de un fichero mediante una **fila**: en *modo de lectura*, el fichero se supone existente previamente a la ejecucion del programa (o bien, si su ubicacion es la terminal, se supone que se crea a la vez que se ejecuta el programa, pero "desde el exterior" de este); en *modo de escritura*, se supone que el fichero no existe, sino que va a ser creado por medio de la **fila** durante la ejecucion del programa.

Operaciones

Las siguientes operaciones predefinidas se aplican a los objetos de tipo **fila**:

- La asignacion y la comparacion *no son posibles* en el caso de las **filas**; consecuentemente, solo es posible un parametro de tipo **fila** si es un **var**-parametro.
- La seleccion de la ventana, que, como hemos visto, se expresa $f\uparrow$, solo es valida si la **fila** esta *conecta*da a un fichero.
- La **accion** predefinida *conecta* toma tres argumentos

  Conecta f,ubicacion,modo;

  (donde *modo* = {*lectura*|*escritura*}) y conecta logicamente una **fila** con un fichero. *F* es la fila a conectar. *Ubicacion* es una (expresion de tipo) tira que establece cual es la ubicacion fisica del fichero; esta tira es dependiente de Sistema Operativo y de Version y no esta definida por las reglas del lenguaje UBL. *Lectura* o *escritura* indican el modo de tratamiento del fichero. *Conecta* es la primera instruccion que debe ejecutarse con un fichero; funciona al modo de la *inicializacion* que debe hacerse con los demas tipos de variable. Despues de ejecutarse *conecta*, $f\uparrow$, si el modo es *lectura*, toma el valor de la primera componente del fichero (si existe); y, si el modo es *escritura*, queda preparado para tomar el valor de lo que puede ser la primera componente del fichero que se esta creando.
- Paralelamente, la **accion**

  Desconecta f;

  termina con la asociacion entre la **fila** y el fichero. $F\uparrow$ pasa a estar indefinido, y *f* queda lista para otra asociacion, si se desea.
- Si el fichero se ha *conecta*do en modo de *lectura*, la **accion**

  Obten f;

  avanza la ventana hasta la siguiente componente del fichero, si esta existe; si no existe, $f\uparrow$ queda indefinido y *fdf(f)* pasa a ser *cierto*. Es incorrecto utilizar *obten* si se verifica *fdf(f)*, o el modo es *escritura*.
- Si el fichero se ha *conecta*do en modo de *escritura*, la **accion**

  Pon f;

  especifica que (el valor de) $f\uparrow$ pasa a formar parte del fichero que se esta creando, y la ventana avanza conceptualmente un espacio, quedando su valor indefinido, de modo que la siguiente operacion *pon* coloque un valor a continuacion de ese. Es incorrecto ejecutar *pon* si el modo es *lectura*.
- La **condicion**

  fdf(f)

indica si la ventana esta mas alla del ultimo elemento valido de la fila. No tiene sentido utilizarla en modo de *escritura*, ya que esto siempre sucede; en modo de *lectura*, indica si la ultima operacion *obten* no obtuvo nada: funciona como si el fichero contuviese un elemento adicional, de valor indefinido, cuya lectura tuviese el efecto de hacer que *fdf(f)* = *cierto*. En ese ultimo caso, *f*↑ esta indefinido.

- La **accion**

  Lee f,v;

  donde *v* ha de ser un objeto variable de tipo *T* si *f* es **fila de** *T*, es una abreviacion util para las instrucciones

  v ← *f*↑; *obten f;*

  y suele utilizarse en vez de estas en la mayoria de los casos.

- Igualmente,

  Escribe f,e;

  donde *e* ha de ser una expresion de tipo *T*, es abreviacion de

  f↑ ← *e*; *pon f;*

El siguiente programa muestra la utilizacion de algunas de las operaciones descritas: calcula la media de los numeros contenidos en un fichero de *enteros*.

```
programa media es -- calcula la media de los elementos de una fila de enteros
  var f: fila de entero; n, suma: entero;
haz
  n ← 0; suma ← 0;
  conecta f,"?????",lectura;
  – "?????" debera substituirse por una tira adecuada, dependiente de version
  mientras no fdf(f) haz
    suma ← suma + f↑;
    n ← n + 1;
    obten f;
  fin mientras;
  desconecta f;
  escribe "La media de los numeros leidos es: ",suma/n;
fin programa;
```

Algoritmo 32. Calcula la media de una fila de enteros

Convenciones de Fin de Fila

Para detectar si se ha llegado a procesar (en modo de *lectura*) el ultimo elemento de un fichero, puede utilizarse la **condicion** predefinida *fdf*, o bien establecer algun tipo de convencion privada: por ejemplo, puede convenirse en que el ultimo elemento tendra un valor valido para su tipo, pero determinado y absurdo para las condiciones del problema: en el Algoritmo 32, podria acordarse que la fila estaria terminada por el numero 9999, y substituir

no *fdf(f)*

por

f↑ = *9999*

A estas convenciones las llamaremos *convenciones de fin de fila*; en un programa, especificaremos siempre cual utilizamos, a menos que nos basemos en la operacion predefinida *fdf*.

Fusion de ficheros

Un problema clasico al estudiar filas es el de *fusionar* dos ficheros. Se suponen los dos ordenados (es decir: cada elemento es mayor o igual que el anterior, con arreglo a determinado criterio de ordenacion), y se quiere obtener un tercer fichero que contenga los elementos de los dos primeros, tambien ordenados. El esquema del siguiente programa resuelve el problema.

```
programa fusion_de_dos_ficheros es

  tipo filaent es fila de entero;

  var f,g: filaent; -- ficheros a fundir; se suponen ordenados
  var h: filaent; -- fichero que contendra el resultado de la fusion

haz

  conecta f,'????',lectura;
  conecta g,'????',lectura;
  conecta h,'????',escritura;

  mientras no fdf(f) y no fdf(g) haz
    si f↑ < g↑ entonces h↑ ← f↑; pon h; obten f; sino h↑ ← g↑; pon h; obten g; fin;
  fin mientras;

  si no fdf(f) entonces
    repite h↑ ← f↑; pon h; obten f; hastaque fdf(f);
  sino
    repite h↑ ← g↑; pon h; obten g; hastaque fdf(g);
  fin si;

  desconecta f;
  desconecta g;
  desconecta h;

fin programa;
```

Algoritmo 33. Fusion de ficheros

Filas de caracteres. El tipo predefinido *texto*

El caso de los ficheros de *caracter*es, que normalmente se utilizan para almacenar textos (en sentido amplio: textos escritos, programas, datos en forma legible, etc.), es mas complejo y requiere de un tratamiento especial. Habitualmente se considera un texto como compuesto de una coleccion de lineas, compuestas a su vez de caracteres; e interesa poder tener en cuenta la estructura de lineas, y no solo la de caracteres. Por todo ello se introduce un tipo predefinido

```
tipo texto es fila de caracter;
```

y se hacen suposiciones adicionales sobre el funcionamiento de, y operaciones aplicables a, los objetos de ese tipo.

Un fichero de texto podra considerarse como una sucesion de lineas; cada linea (al ser examinada o creada; y automaticamente, como se vera) contendra un caracter adicional a la derecha, llamado *caracter de fin de linea*, que "funcionara" como si fuese un espacio en blanco al posicionarse la ventana encima.

```
esta es la primera linea del fichero$
de caracteres$
     $
la linea anterior contiene exactamente$
cinco espacios$
```

Figura 74. Un fichero de texto: El caracter de fin de linea se representa mediante el caracter "$".

Este caracter tiene el efecto adicional de que, al ser examinado mediante la ventana, la condicion predefinida *fdl(f)* pasa a valer *cierto* (y solo lo hace en esos casos).

Las operaciones aplicables a los objetos de tipo *texto* (ademas de las que se aplican a todos los de tipo fila) son

- La condicion *fdl* (Fin De Linea)

  condicion *fdl(***var** *f: Texto);*

  que vale *cierto* si y solo si la ventana esta posicionada sobre el caracter de fin de linea. En ese caso, y por convencion, el valor de la ventana es un espacio en blanco (' ').

- En modo de *lectura*, la

  accion *lee_linea(***var** *f: Texto);*

  avanza la ventana hasta despues del primer caracter de fin de linea que encuentre, lo cual equivale logicamente a pasar a principio de la linea siguiente (si existe, y no hay fin de fila).

- En modo de *escritura*, la

  accion *escribe_linea(***var** *f: Texto);*

  asigna un caracter de fin de linea a la ventana y lo *pone* en el fichero, cambiando efectivamente de linea y empezando una nueva. Este es el unico modo de escribir un caracter de fin de linea desde un programa.

Ademas, para filas de tipo *texto* son validas las siguientes convenciones

- A diferencia de los demas tipos fila, para los cuales es obligatoria la coincidencia de tipos entre el objeto o expresion que se *lee* o *escribe* y la ventana, para objetos de tipo *texto* es posible utilizar objetos o expresiones de tipo *entero, real, logico, tira*, enumerado o subrango: en cualquiera de estos casos, se *lee* o *escribe* una sucesion de caracteres que forma una tira que contiene un valor de uno de esos tipos, de acuerdo con la sintaxis especificada por el lenguaje:

  por ejemplo,

  n ← 54; escribe t,n;

  y

  escribe t,'5'; escribe t,'4';

  tienen el mismo efecto.

  Se notara que esta definicion respeta y coincide con los metodos de lectura y escritura que se han utilizado informalmente en los capitulos anteriores.

- Una instruccion como

  escribe $f,v_1,v_2,\ldots,v_n$;

  donde f es un *texto* y v_i expresiones, se considera abreviacion de

  escribe f,v_1; *escribe* f,v_2; ... *escribe* f,v_n;

  y similarmente para la instruccion *lee*.

- Se admiten parametros adicionales en las instrucciones *lee_linea* y *escribe_linea*:

  lee_linea f,o_1*,*o_2*,....*o_n*;*

  se considera equivalente a

  lee f,o_1*,*o_2*,...,*o_n*; lee_linea f;*

  y lo mismo con *escribe_linea* y *escribe.*

- Las lecturas y escrituras con formato se comprenden como extensiones de las anteriores reglas.

- Por ultimo, se consideran predefinidos los objetos

  var *entrada, salida: Texto;*

  y se considera que, antes de la ejecucion de la primera instruccion del programa, se ejecuta

  conecta entrada,'?????',lectura;
  conecta salida,'?????',escritura;

  prohibiendo la ejecucion de cualquier instruccion *conecta* o *desconecta* sobre esos objetos. Ademas, si la fila es *entrada* en *fdf, fdl, lee o lee_linea*, puede omitirse, igual que *salida* es *escribe* o *escribe_linea.*

```
programa copia es haz
  mientras no fdf haz
    mientras no fdl haz
      salida↑ ← entrada↑;
      obten entrada; pon salida;
    fin mientras;
    escribe_linea; lee_linea;
  fin mientras;
fin programa;
```

Algoritmo 34. Copia la fila de texto predefinida entrada en la fila de texto predefinida salida: Notese el uso de abreviaciones: *fdl* significa *fdl(entrada)*, etc...

Control de Stocks

El siguiente programa actualiza un fichero de stocks. Para cada articulo, el fichero (que llamaremos *fichero maestro*) contiene un numero que lo identifica, un descripcion de como maximo 40 caracteres, y un numero que indica su cantidad (se supone el fichero maestro ordenado respecto de el numero de identificacion). Cada dia, se lleva control de los articulos que se compran y venden, de modo que se genera un *fichero de actualizaciones.* Se distinguen entre tres tipos de actualizacion, identificados mediante una clave alfabetica: añadir (A) un articulo al stock, suprimirlo (S) o cambiarlo (C) -- ya sea en la cantidad o en su descripcion. Cada dia se ordena por numero de articulo el fichero de actualizaciones, y se ejecuta el siguiente programa, que compone un nuevo fichero maestro (o *de resultados*) y, eventualmente, un *fichero de error*, con listado de las actualizaciones que no han podido realizarse, por ser erroneas. La convencion de fin de fila esta explicada en el programa.

```
programa actualizacion es
  tipo nombre_articulo es :tira(40);
  tipo ficha es tupla numero: entero; descripcion: nombre_articulo; cantidad: entero; fin;
  tipo operacion es {A,S,C}; -- Añadir, Cambiar, Suprimir
  var entrada,salida,entrada2,salida2: texto; fm,fa: ficha; op: operacion;
  (* convencion de fin de fila: la ultima ficha de los ficheros maestro y de actualizacion *)
  const fin_de_fila = 999999; (* indican el fin de fila al tener como numero 999999 *)

  accion lee_maestro (var f: ficha) haz
    lee_linea entrada,f.numero:10,f.descripcion:40,f.cantidad:10;
  fin lee_maestro;

  accion lee_actualizacion (var op: operacion; var f: ficha) haz
    lee entrada2,op:1,f.numero:10;
    si op en {C,A} entonces lee entrada2,f.descripcion:40,f.cantidad:10; fin;
    lee_linea entrada2;
  fin lee_actualizacion;

  accion pon_maestro (const f: ficha) haz
    escribe_linea salida,f.numero:10,f.descripcion,f.cantidad:10;
  fin pon_maestro;

  accion pon_error (const op:operacion; const f: ficha) haz
    escribe_linea salida2,op:1,f.numero:10,f.descripcion,f.cantidad:10;
  fin pon_error;

haz
  conecta entrada,'????',lectura; conecta entrada2,'????',lectura;
  conecta salida,'????',escritura; conecta salida2,'????',escritura;
  lee_maestro fm; lee_actualizacion op,fa;
  repite
    mientras fm.numero < fa.numero haz pon_maestro fm; lee_maestro fm; fin;
    si fm.numero = fa.numero entonces -- Cambiar, Suprimir
      segun op es
        cuando A => pon_error op,fa; lee_actualizacion op,fa;
        cuando C =>
          repite
            fm ← fa; lee_actualizacion op,fa;
          hastaque op ≠ C o fm.numero ≠ fa.numero;
        cuando S => lee_maestro fm; lee_actualizacion op,fa;
      fin segun;
    sino -- Añadir
      repite
        si op = A entonces pon_maestro fa; sino pon_error op,fa; fin;
        lee_actualizacion op,fa;
      hastaque fa.numero ≥ fm.numero;
    fin si;
  hastaque fm.numero = fin_de_fila o fa.numero = fin_de_fila;
  decide
    cuando fa.numero ≠ fin_de_fila =>
      repite
        si op = A entonces pon_maestro fa; sino pon_error op,fa; fin;
        lee_actualizacion op,fa;
      hastaque fa.numero = fin_de_fila;
    cuando fm.numero ≠ fin_de_fila =>
      repite
        pon_maestro fm; lee_maestro fm;
      hastaque fm.numero = fin_de_fila;
  fin decide;
  pon_maestro fm;
  desconecta entrada; desconecta entrada2; desconecta salida; desconecta salida2;
fin programa;
```

Algoritmo 35. Control de Stocks

Subprogramas como parametros

Ademas de objetos (pasados por copia, por variable y por constante), un subprograma puede admitir entre sus parametros lo que llamaremos *subprogramas formales*; este tipo de parametro se utiliza para diseñar subprogramas de algun modo *genericos*, en el sentido de que, dependiendo del *subprograma actual* pasado como argumento en la invocacion, actuaran (o realizaran evaluaciones; o trataran secuencias) de formas distintas.

---

parametros_por_subprograma = cabecera

Figura 75. Sintaxis de un parametro por subprograma

Son ejemplos de subprogramas con parametros por subprograma

accion *A (***accion** *B;* **funcion** *H: real);*
funcion *F (x: real;* **funcion** *G(z: real): real): entero;*
secuencia *S (***secuencia** *T: real): real;*

Una lista de parametros puede contener parametros por subprograma de cualquier tipo, arbitrariamente mezclados con parametros por copia o por nombre. Las cabeceras que declaran los parametros por subprograma pueden contener a su vez parametros; los nombres de estos "parametros de segundo orden" (p. ej., el parametro *z* de la funcion parametrica *G* de la funcion *F*) pueden ser arbitrarios, y no sirven mas que para indicar el tipo de los parametros del subprograma parametrico.

Los subprogramas actuales pasados como argumento deben tener listas de parametros *compatibles* con las de los parametros declarados, en el sentido de que los tipos de sus parametros deben ser identicos y estar en el mismo orden (o, si se da el caso, los parametros por subprograma deben ser compatibles), aunque los identificadores de los parametros difieran.

Por ejemplo, una

funcion *F (x: real;* **funcion** *G(z: real): real): entero;*

puede ser invocada como

F(3.14,h)

y producira un valor entero, siempre que la declaracion del argumento o subprograma actual *h* tenga una lista de parametros compatible con la de *G*; por ejemplo, si esta declarada como

funcion *h(r: real): real*

(Notese que el parametro de *h* se llama *r* y el de *G* se llama *x*; esto no tiene importancia, ya que los dos son de tipo *real*).

Integracion por trapecios

Una de las formas mas simples de integrar numericamente una

funcion *f(x: real): real;*

entre los valores *a* y *b* es el *metodo de los trapecios*:

Se divide el intervalo *[a,b]* en trozos convenientemente pequeños:

$x_0 = a,\ x_1 = a + \Delta,\ x_2 = a + 2*\Delta,\ \ldots\ x_{n-1} = b - \Delta,\ x_n = b$

Para cada intervalo $[x_i, x_{i+1}]$, se aproxima la integral sobre ese intervalo como el area del trapecio formado por los puntos

$x_i,\ x_{i+1},\ f(x_i),\ f(x_{i+1})$

que vale

$(f(x_i) + f(x_{i+1}))*\Delta/2$

La integral completa se evalua como la suma de las integrales aproximadas de los intervalos, es decir, como

$I = \Delta \times ((f(x_0)+f(x_n))/2 + f(x_1) + f(x_2) + \ldots + f(x_{n-1}))$

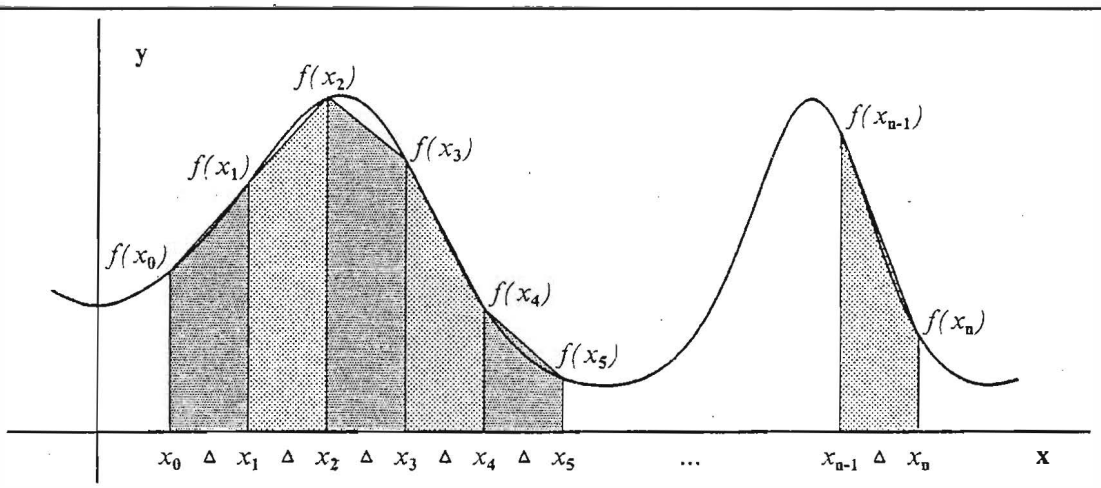


Figura 76. Ejemplo de integracion por trapecios.

La siguiente funcion implementa ese metodo:

```
funcion integral(funcion f(x:real): real; a,b: real; n: entero): real es
  var i,i2,i1,delta,Xi: real; k: entero;
haz
  i1 ← f(a) + f(b);
  delta ← (b-a)/n;
  Xi ← a + delta;
  i2 ← 0;
  para k en asc(1,n-1) haz
    i2 ← i2 + f(Xi);
    Xi ← Xi + delta;
  fin para;
  i ← delta*(i1/2 + i2);
  vale i;
fin integral;
```

Algoritmo 36. Integracion numerica por el metodo de los trapecios

y se utiliza, dada una funcion y declaraciones como

```
var x,z: real; n: entero;
funcion cubo(x: real): real haz vale x*x*x; fin;
```

en formas del estilo de

```
escribe_linea "La integral de F(x) = x*x*x entre los puntos ",
  x," e ",y," utilizando ",n," puntos es: ",integral(cubo,x,y,n);
```

Filtros de secuencias

Dada cualquier secuencia S de numeros enteros, y una propiedad P sobre los enteros, el siguiente subprograma implementa otra secuencia, que contiene solo los elementos de S que verifican P

```
secuencia filtro(secuencia S: entero; condicion P(n: entero)): entero es
  var n: entero;
haz
  para n en S talque P(n) haz produce n; fin;
fin filtro;
```

Algoritmo 37. Un filtro para secuencias de enteros

El metodo es general, y da una idea de como escribir filtros abstractos arbitrarios que restrinjan secuencias segun cualquier criterio.

Recursividad

Introduccion

Hasta ahora hemos estudiado diversos tipos de subprograma, y visto como cualquiera de ellos puede invocar a cualquier otro (siempre que su nombre sea visible en el punto de la invocacion); lo que no hamos visto es que pasa si un subprograma se invoca (directa o indirectamente) a si mismo. Cuando sucede esto ultimo, se dice que el subprograma es *recursivo*, y que la invocacion que de si mismo hace es una *invocacion recursiva*. El lenguaje trata este tipo de invocaciones sin inconvenientes, aunque se plantean algunos problemas, que intentaremos aclarar en esta discusion, introduciendo cuando sea preciso nuevos conceptos y definiciones.

El subprograma

```
accion recursiva(i: entero) es
  si i > 0 entonces
    recursiva i div 10;
    escribe caracter(ordinal('0') + i mod 10);
  fin si;
fin recursiva;
```

escribe todo entero mayor que 0 en su representacion en forma de caracteres:

En efecto, una invocacion como *recursiva 12;* se ejecuta del siguiente modo:

1. En primer lugar, se invoca *recursiva 1;* que, a su vez se ejecuta como sigue:
   a. Se invoca *recursiva 0;* (cuyo efecto sera nulo).
   b. Y a continuacion se escribe el caracter '1'.
2. En segundo lugar, se escribe el caracter '2'

con lo que el efecto resultante es el descrito (como lo es tambien para cualquier otro argumento, como puede deducirse del examen del subprograma).

Este no es por si mismo un buen ejemplo de uso de recursividad, ya que el mismo efecto podria haber sido obtenido (ademas, de un modo mas claro) mediante una iteracion (cuya escritura se plantea como ejercicio); sin embargo, muestra el mecanismo de la recursividad, y nos permite plantear algunas cuestiones.

- Cada invocacion (recursiva o no) de un subprograma crea una nueva *instancia* o copia de ese subprograma;
- en cada instancia existe una copia distinta de los objetos locales declarados en ese subprograma (en nuestro ejemplo, la *i* de la segunda invocacion [recursiva] es *distinta* de la *i* de la primera invocacion); por tanto,
- es imposible acceder desde una instancia de un subprograma a los objetos declarados en otra instancia del mismo subprograma (no hay peligro de confusion entre las *i*s, ya que en la segunda instancia solo podemos manipular la alli declarada).

Como ya se ha apuntado, no siempre es conveniente dar soluciones recursivas a los problemas; en particular, cuando existen versiones iterativas conocidas.

El esquema

```
mientras C haz I; fin;
```

puede ser reescrito como invocacion a una accion recursiva *R* cuyo cuerpo es

```
si C entonces I; R; fin;
```

pero tal reescritura es menos clara y menos eficiente que su contrapartida iterativa.

La recursividad puede considerarse como una consecuencia de la metodologia de diseño descendente: cada abstraccion se refina en funcion del vocabulario existente, y por tanto es posible utilizar alguna abstraccion en su propio refinamiento. Ello nos da la clave de la recursividad: puesto que cada refinamiento consiste en expresar una tarea en forma de varias sub-tareas "mas pequeñas", no hay problema en que las sub-tareas vengan descritas por el mismo subprograma, siempre que

- sean efectivamente sub-tareas, es decir, en cierto sentido "mas pequeñas" o "que den menos trabajo"; y que "tiendan" a
- la (o las) subtarea(s) "mas pequeñas", que seran resueltas sin la ayuda de la recursion.

Llamaremos grafo o *arbol de invocaciones* a la representacion arborea de las instancias producidas por una serie de invocaciones. Por ejemplo, el grafo de invocaciones de

recursiva 154;

sera, representando en cada nodo el correspondiente argumento,

recursiva ——> *recursiva* ——> *recursiva* ——> *recursiva*
154 *15* *1* *0*

y el efecto se seguira del recorrido inverso del grafo

escribe 4 <—— *escribe 5* <—— *escribe 1* <—— **nada**;

Para reproducir de otro modo las condiciones del parrafo anterior: ha de garantizarse la finitud del grafo para cualesquiera argumentos del subprograma.

Recursion directa e indirecta. Definiciones e implementaciones de subprogramas

Un subprograma es *directamente recursivo* cuando se invoca a si mismo en alguna de las instrucciones de su cuerpo, e *indirectamente recursivo* cuando lo hace a traves de otro(s) subprograma(s):

```
accion A es
  accion B haz ... A; ... fin;
haz
  ...
  B; -- recursion indirecta
  ...
fin A;
```

Si los subprogramas estan declarados en el mismo nivel declarativo o son *mutuamente recursivos* (esto es, *A* invoca a *B* y *B* invoca a *A*) puede ser necesario el uso de "predeclaraciones" o *definiciones* de los subprogramas (que consisten en escribir su cabecera seguida de un punto y coma) separadas de sus respectivas *implementaciones* (que contienen el cuerpo del programa).

Si *A* invoca a *B* y *B* invoca a *A*, no hay ningun orden de declaracion de *A* y *B* que satisfaga la regla de "declaracion antes de uso", a menos que utilizemos definiciones e implementaciones:

```
accion A; -- definicion de A

accion B es ... haz ... A; ... fin; -- implementacion de B.
-- Usa A, puesto que ya esta definido, pero debe darse su implementacion

accion A es ... haz ... B; ... fin;
-- implementacion de A. En este caso B esta tambien implementado, ademas de definido.
```

La sucesion de Fibonacci

La sucesion de Fibonacci

f: *1 1 2 3 5 8 13 21 34 55 89 ...*

se forma partiendo de los dos primeros elementos, que valen 1, por el metodo de hallar cada elemento como suma de los dos anteriores. Una formula (recursiva) que describe la sucesion *f* puede ser

$f_0 = f_1 = 1$
$f_n = f_{n-1} + f_{n-2}$ *para* $n > 1$

y permite implementar rapidamente un algoritmo para calcular cualquiera de sus terminos:

```
funcion Fibonacci(i: entero): entero haz
  si i = 0 o i = 1 entonces vale 1;
  sino vale Fibonacci(i-1) + Fibonacci(i-2);
  fin si;
fin Fibonacci;
```

Algoritmo 38. La sucesion de Fibonacci

Desde el punto de vista de la eficiencia, el metodo es desastroso, como puede verse en el siguiente arbol de invocaciones, donde en cada punto se ha escrito solo el valor del parametro:

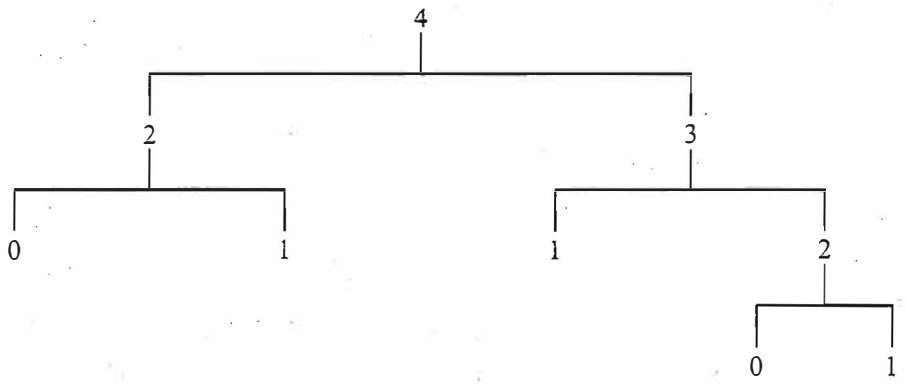


Figura 77. Arbol de invocaciones para Fibonacci(4): Cada camino se detiene al llegar a I=0 o I=1.

y debe ser substituido por un esquema iterativo:

```
funcion Fibonacci(i: entero) es
  var f,fant: entero;
haz
  si i = 0 o i = 1 entonces vale 1;
  sino
    f ← 1; fant ← 1;
    repite
      f ← f + fant; fant ← f - fant;
      i ← i - 1;
    hastaque i = 1;
    vale f;
  fin si;
fin Fibonacci;
```

Algoritmo 39. La sucesion de Fibonacci, version iterativa

Las torres de Hanoi

Se dispone de tres estacas, llamadas *A*, *B* y *C*, y de una coleccion de discos de anchuras distintas entre si, perforados de modo que puedan ensartarse en las estacas. El problema de las Torres de Hanoi se plantea como sigue:

> En la estaca *A* se disponen *n* discos, ordenados de mayor a menor, de modo que el menor este encima. Se trata de trasladar los discos de la estaca *A* a la estaca *B*, usando la *C* como auxiliar si es preciso, dejandolos en el mismo orden en que se encuentran, y utilizando cualquier numero de veces las siguientes reglas de movimiento:
>
> - En cada movimiento se puede cambiar solo un disco de estaca; ese disco debe de estar encima en la estaca de partida, e ira a parar encima en la estaca de llegada.
> - Ninguno de los movimientos debe llevar a una situacion en la que un disco grande descanse sobre uno mas pequeño.

y tiene solucion, como se demuestra facilmente por induccion:

- Para $n = 0$ trivialmente hay una solucion, ya que sin hacer nada queda resuelto el problema.
- Supuesto resuelto el problema para *n-1* discos, el problema con *n* discos se resuelve del siguiente modo:
  1. Se trasladan *n-1* discos de la estaca *A* a la estaca *C* (lo cual es posible por la hipotesis de induccion).
  2. Seguidamente, se mueve el ultimo disco de la estaca *A* a la *B* (lo cual es posible, ya que no contraviene ninguna de las reglas prescritas).
  3. Por ultimo, se trasladan los *n-1* discos restantes de la estaca *C* a la *B*, con lo cual queda ultimada la prueba de existencia de solucion para *n* discos.

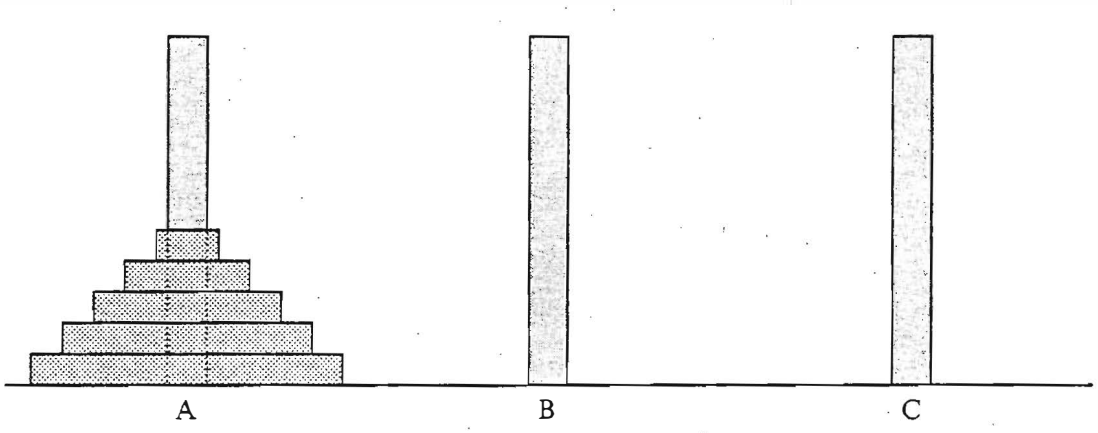


Figura 78. Las Torres de Hanoi: posicion inicial con 5 discos

De la demostracion se deduce directamente el siguiente programa

```
programa Torres_de_Hanoi es

  tipo Torre es {A,B,C};

  accion Hanoi(partida,llegada,auxiliar: torre; n: entero) haz
    si n ≠ 0 entonces
      Hanoi partida,auxiliar,llegada,n-1;
      escribe_linea "Mover de ",partida," a ",llegada,".";
      Hanoi auxiliar,llegada,partida,n-1;
    fin si;
  fin Hanoi;

  var n: entero;

haz
  lee n;
  Hanoi A,B,C,n;
fin programa;
```

Algoritmo 40. Las Torres de Hanoi

cuya singularidad radica en la ausencia de estructuras de datos que representen las estacas o los discos. El siguiente esquema muestra graficamente los movimientos necesarios para $n = 2$.

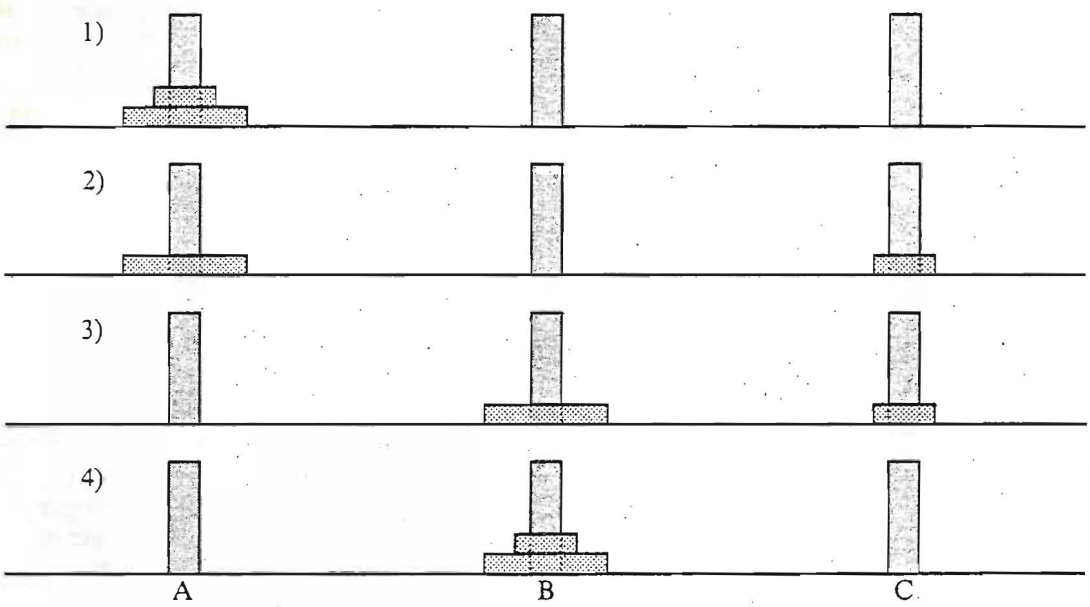


Figura 79. *Movimientos para el caso de dos discos:*

1. Posicion inicial
2. Primer movimiento: de *A* a *C*
3. Segundo movimiento: de *A* a *B*
4. Tercer y ultimo movimiento: de *C* a *B*

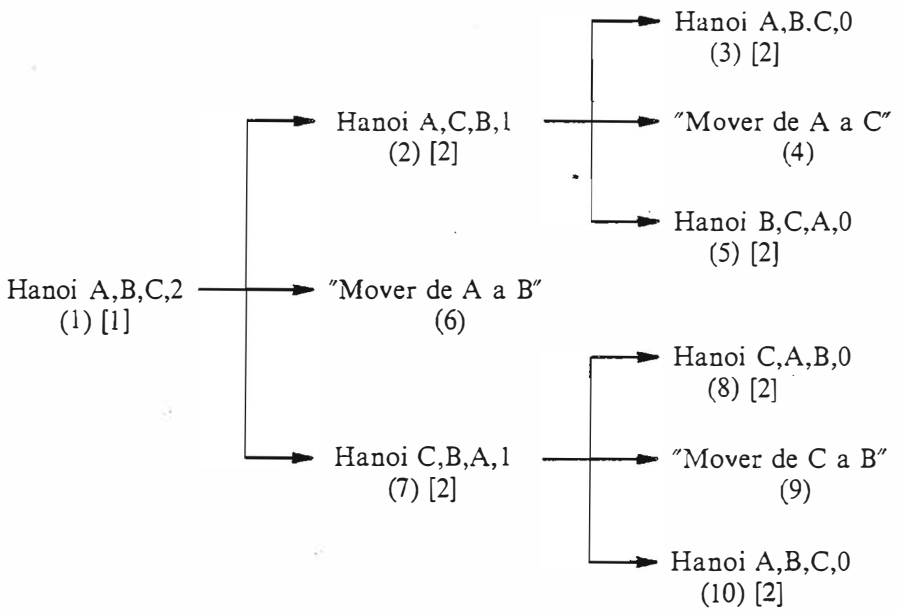


Figura 80. ***Grafo de invocaciones para Hanoi A,B,C,2:*** Los numeros entre parentesis indican el orden en el que se ejecutan las instrucciones; los que estan entre corchetes, el numero de instancias activas de la **accion** *Hanoi*. Las invocaciones cuyo cuarto argumento es 0 no se han expandido, ya que su efecto es nulo.

Descomposicion de un entero en suma de dados

Dado un entero positivo p, se trata de descomponerlo de todos los modos posibles como suma de (valores de caras de) dados, sin importar el orden en que se sacan los valores (esto es, considerando equivalente $3=2+1$ y $3=1+2$). La ultima propiedad permite elegir una representacion conveniente de las descomposiciones, por ejemplo

$$p = n_6*6 + n_5*5 + n_4*4 + n_3*3 + n_2*2 + n_1*1$$

donde n_i significa "numero de dados con valor i".

Una aproximacion sistematica al problema permite ver que n_6 podra tomar cualquier valor entre 0 y p **div** 6, quedando condicionados los demas n_i por la eleccion que se haga a descomponer el valor $p - 6*n_6$; el mismo razonamiento, aplicado a ese valor, permite obtener, para cada eleccion de n_6, un conjunto de elecciones posibles para n_5, y un nuevo resto

$$p - 6*n_6 - 5*n_5$$

y asi, hasta llegar a

$$p - 6*n_6 - 5*n_5 - 4*n_4 - 3*n_3 - 2*n_2$$

que esta determinado y se obtiene directamente, sin eleccion.

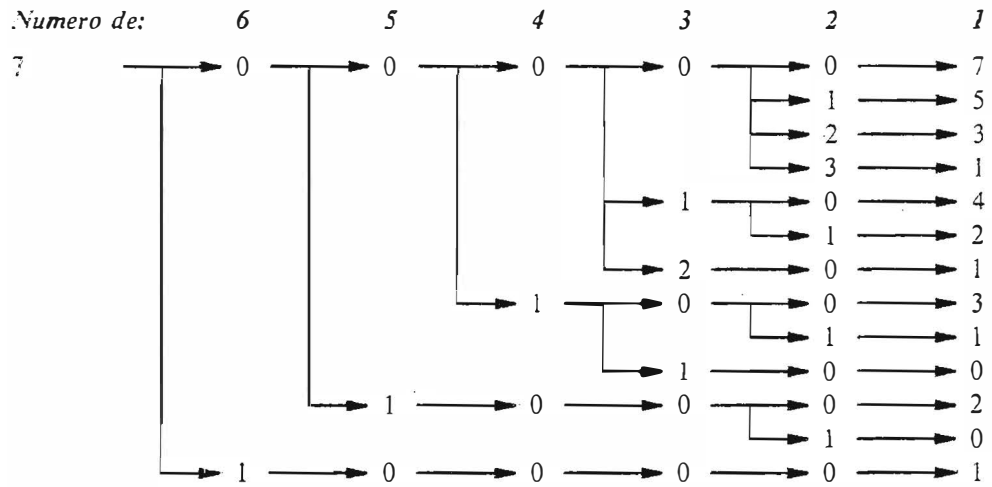


Figura 81. Descomposiciones posibles del numero 7

Del analisis mostrado se deriva casi directamente el siguiente programa, en el que se han representado las n_i mediante una **tabla**

```
programa Dados es

  var Pp: entero;
  var N tabla [1..6] de entero;

  accion descompon(p,i: entero) es

    var k: entero;

    accion solucion es
      var j: entero; primer_ni: logico;
    haz
      primer_ni ← cierto;
      escribe Pp," = ";
      para j en desc(6,i) talque N[j] ≠ 0 haz
        si primer_ni entonces primer_ni ← falso; sino escribe " + "; fin;
        escribe N[j],'*'j;
      fin para;
      escribe_linea;
    fin solucion;

  haz
    si i = 1 entonces N[1] ← p; solucion;
    sino
      para k en asc(0,p div i) haz
        N[i] ← k;
        si k*i = p entonces solucion; sino descompon p - k*i,i-1; fin;
      fin para;
    fin si;
  fin descompon;

haz
  lee Pp; descompon Pp,6;
fin programa;
```

Algoritmo 41. Descomposicion en suma de dados

Las Ocho Damas, en version recursiva

El problema de las Ocho Damas (vease el Algoritmo 27 en la pagina 102) admite tambien una formulacion recursiva. Basta con considerarlo como consistente en "probar" de poner las ocho damas, dando en este caso a "probar" un significado recursivo: "probar de poner una dama" (en una fila dada) sera "poner una dama en cada posicion no conflictiva" (de su fila) seguido cada vez de "probar de poner una dama" (en la siguiente fila). De este razonamiento se deriva el siguiente programa, que resulta ser mucho mas simple y corto que su correspondiente version iterativa.

```
programa Ocho_damas es

  var da: aplicacion [-7..7] => logico;
  var dd: aplicacion [2..16] => logico;
  var col: aplicacion [1..8] => logico;
  var D: tabla [1..8] de [1..8];

  accion inicializa es var i: entero; haz
    para i en asc(-7,7) haz da[i] ← falso; fin;
    para i en asc(2,16) haz dd[i] ← falso; fin;
    para i en asc(1,8) haz col[i] ← falso; fin;
  fin inicializa;

  accion solucion es var i: entero; haz
    para i en asc(1,8) haz escribe ' ',D[i]; fin; escribe_linea;
  fin solucion;

  accion prueba(i: entero) es
    var j: entero;
  haz
    para j en asc(1,8) talque no(col[j] o da[i-j] o dd[i+j]) haz
      D[i] ← j;
      col[j] ←cierto; da[i-j] ←cierto; dd[i+j] ←cierto;
      si i = 8 entonces solucion; sino prueba i+1; fin;
      col[j] ← falso; da[i-j] ← falso; dd[i+j] ← falso;
    fin para;
  fin prueba;

haz
  inicializa; prueba 1;
fin programa;
```

Algoritmo 42. Las Ocho Damas en version recursiva

Ejercicios

1. Reprogramar la **accion** *recursiva* de la introduccion en forma iterativa.

2. Justificar en el Algoritmo 41 en la pagina 133 el uso de la variable logica *primer_ni* y la posicion de la **accion** *solucion*.

3. La *funcion de Ackermann* se define como

   $A(0,n) = n+1$, para $n \geq 0$
   $A(m,0) = A(m-1,1)$, para $m > 0$
   $A(m,n) = A(m-1,A(m,n-1))$, para $m,n > 0$

   escribir dos algoritmos, uno recursivo y otro iterativo, para calcular $A(m,n)$. Estudiar el grafo de invocaciones para algunos valores de m y n, por ejemplo 2,2.

Nombres

Los nombres (llamados *apuntadores* o *pointers* en otros lenguajes) son objetos cuyo valor es el nombre de otro objeto.

Figura 82. Un objeto de tipo **nombre** *y el objeto nombrado:* En este caso, el tipo de *n* es **nombre** *entero*. *β* es un nombre interno, no accesible por el programa de otro modo que como valor de *n* (o como valor de otro objeto del mismo tipo que *n*). El objeto referenciado por *n* (esto es, *β*) puede tratarse mediante la notacion *n*↑. La flecha dibujada en la figura indica la relacion conceptual que existe entre los dos objetos: por esta razon se ha llamado a veces a los **nombres** "apuntadores".

Los objetos o campos de tipo **nombre** se declaran siempre mediante un identificador de tipo, previamente asociado a un tipo **nombre** mediante una declaracion del tipo

tipo *identificador* **es nombre** *tipo;*

Figura 83. Sintaxis de una declaracion de tipo **nombre**

Por ejemplo, el objeto *n* de la Figura 82 puede declararse como

```
tipo T es nombre entero;
var n: T;
```

pero no como

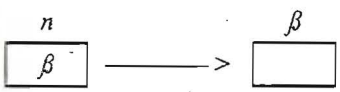
```
var n: nombre entero; -- incorrecto
```

El reino de los valores de un objeto de tipo **nombre** incluye nombres (como *n*) de objetos (como *n*↑), como se ha visto, ademas del valor **nulo** (**nulo** es un identificador reservado), que puede entenderse como "el nombre de ningun objeto"; si *n* vale **nulo**, es un error hablar de *n*↑. En cuanto a los valores que no son **nulo**, la unica forma de dar valor a un objeto de tipo **nombre**, aparte de la asignacion, es utilizando la **accion** predefinida *crea*:

```
crea n;
```

por ejemplo, tiene el efecto de

- *Crear* un objeto del tipo del cual *n* es **nombre** (esto es, *entero*), y
- asignar su nombre como valor de *n*.



El objeto creado, *n*↑, esta indefinido hasta que se le de valor:

$n\uparrow \leftarrow -51$

y, a diferencia de los objetos declarados como **variables**, no ve limitada su existencia por ningun ambito sintactico. Puede ser destruido mediante la instruccion

libera n;

cuyo efecto es destruir la variable $ß$, con lo que $n\uparrow$ deja de ser valido.

Dada una declaracion como

var *i: entero;*

podria pensarse en asignar

$n \leftarrow i$; -- *incorrecto*

con la intencion de que *n* "apuntase" a *i*; sin embargo, tal asignacion es incorrecta, y como tal es señalada por el compilador: los tipos de *n* (esto es, **nombre** *entero*) e *i* (esto es, *entero*) no son compatibles. Lo que si puede hacerse sin problema es

$n\uparrow \leftarrow i$;

En resumen: es imposible hacer que un objeto de tipo **nombre** "apunte" a un objeto declarado localmente. De otro modo: el reino de los valores de los objetos de tipo **nombre** y el reino de los nombres de objetos locales son disjuntos.

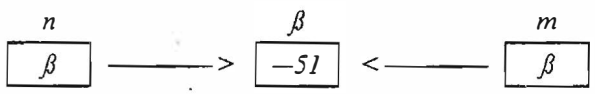
Ademas, si suponemos declarado

var *m: T;*

es posible asignar

$m \leftarrow n$;

con lo que se llega a la situacion representada en la siguiente figura:



Dos objetos de tipo **nombre** pueden "apuntar" pues al mismo objeto.

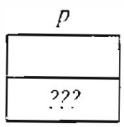
Principal uso de los objetos de tipo nombre

El interes de los **nombres** puede parecer limitado, al añadir complicaciones adicionales al manejo de variables, pero se hace evidente al combinar **nombres** con la nocion de **tupla** y la posibilidad de definir *tipos indirectamente recursivos*.

Obviamente, un tipo como

```
tipo erroneo es
  tupla
    a: entero;
    b: erroneo;
  fin tupla;
```

es incorrecto; para verlo, basta con pensar en el aspecto de un objeto *p* de tipo *erroneo*:



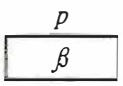
campo a, *de tipo* entero

campo b, *de tipo* erroneo: *cual es su forma?*

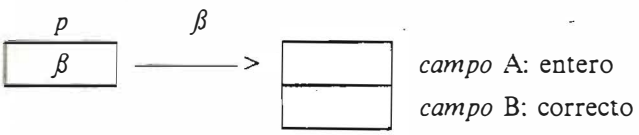
Diremos que un tal tipo es *directamente recursivo*, y prohibiremos su uso y declaracion; sin embargo, si declaramos

```
tipo correcto es nombre
  tupla
    a: entero;
    b: correcto;
  fin tupla;
```

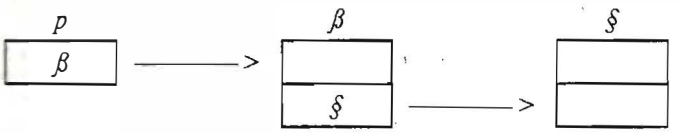
utilizando en este caso *recursion indirecta*, no hay ningun problema al considerar la forma de un objeto *p* de tipo *correcto*:



β sera, a su vez, o bien el valor **nulo**, o bien el nombre interno de una **tupla** con los componentes descritos:



De modo que en ese caso podra hablarse de $p\uparrow.a$ para referirse al campo *entero* de la **tupla**, y de $p\uparrow.b$ para hablar de su campo *correcto*. Por su parte, el valor del campo $p\uparrow.b$ podra tambien ser **nulo**, o bien el nombre de otra tupla (dado que su tipo es *correcto*):



Lo cual permitira referirse a $p\uparrow.b\uparrow.a$ y $p\uparrow.b\uparrow.b$ para hablar de los campos de la nueva tupla. Y asi sucesivamente.

Extendiendo el razonamiento se observa que *p* puede dar acceso a una *lista* completa de tuplas; ademas, y puesto que al valor de *p* (o del campo *b* de cualquier tupla) puede ser **nulo**, se ve tambien que el numero de componentes de esa lista es manipulable (ya que la presencia del valor **nulo** indica, por convencion, fin de lista) [de hecho, los valores a que da acceso *p* no tienen porque formar una lista, sino que pueden seguir una estructura de bucle].

En resumen: la declaracion de un unico objeto de tipo **nombre** permite el acceso y la manipulacion de una lista arbitrariamente larga de (tuplas que contienen, en este caso) enteros. En general, y considerando la posibilidad de declarar mas campos, es posible manipular grafos arbitrariamente complejos. Estudiaremos aqui algunos de los mas utilizados en programacion.

Listas unidireccionales

La forma mas sencilla de encadenar un conjunto de elementos es ponerlos en una *lista unidireccional* (o simplemente *lista*).

```
tipo lista es nombre
  tupla
    siguiente: lista; -- resto de la lista
    e: entero; -- elemento
  fin tupla;
```

El tipo *entero* se utiliza unicamente como ejemplo; la teoria expuesta se generaliza sin dificultad para otros tipos. En el resto de este parafo, se considera declarado

var *p: lista;*

La representacion de (un posible valor de) *p* sera

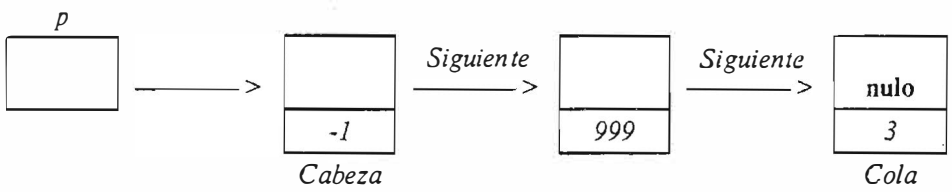


Figura 84. Una lista unidireccional de enteros

Llamaremos *cabeza* al principio de la lista (esto es, al valor de *p*) y *cola* a su final. Diremos que la lista esta *ordenada* si lo estan su componentes (enteras) al reseguir la lista desde la cabeza hasta la cola. Para *insertar* un elemento *q* de valor *v* en la lista por su cabeza, basta con hacer

```
crea q; -- crea un objeto
con q↑ haz siguiente ← p; e ← v; fin; -- encadenalo en la lista
p ← q; -- modifica el valor de la cabeza
```

Observese que el metodo vale para todos los casos, incluido aquel en que la lista es vacia, o, lo que es lo mismo, *p* = **nulo**.

Para *insertar* un elemento de nombre *q despues* de una posicion arbitraria de nombre *r*, bastara con seguir el metodo indicado en la siguiente figura:

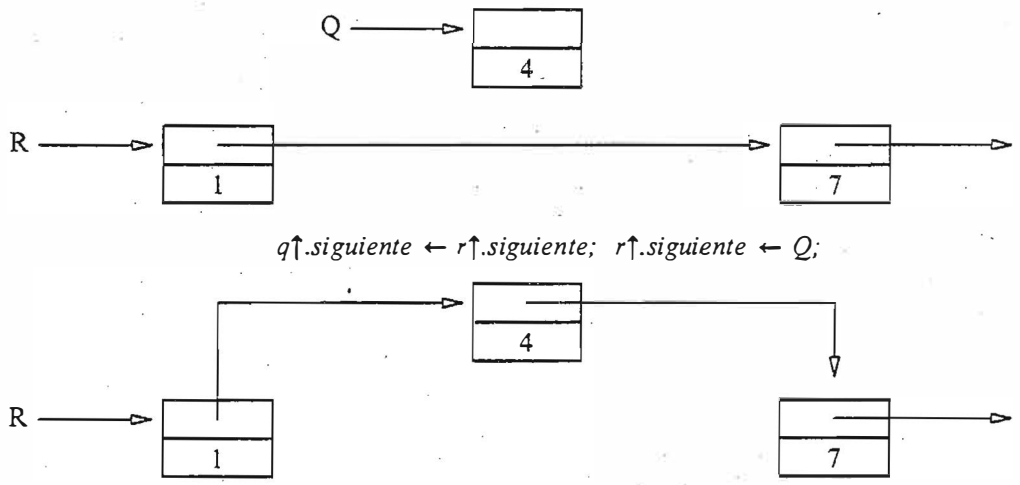


Figura 85. Insercion en una lista unidireccional

La *supresion* del *siguiente* de un elemento de nombre *q* es igualmente sencilla,

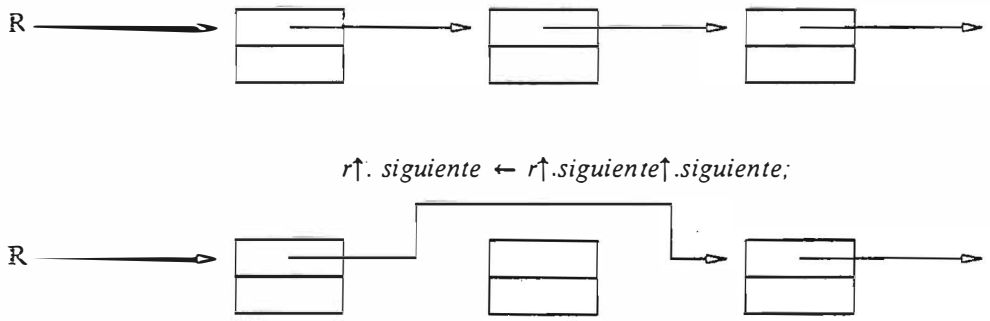


Figura 86. Supresion del siguiente de un elemento en una lista unidireccional

pero no lo es tanto la subresion del propio $q\uparrow$, que se plantea como ejercicio.

Es trivial diseñar **secuencias** para tratar los elementos de una lista:

```
-- genera nombres de los elementos de la lista
secuencia elem(l: lista): lista haz
  mientras l ≠ nulo haz
    produce l;
    l ← l↑.siguiente;
  fin mientras;
fin elem;

-- genera los valores de los elementos de la lista
secuencia val(l: lista): entero haz
  mientras l ≠ nulo haz
    produce l↑.e;
    l ← l↑.siguiente;
  fin mientras;
fin val;
```

Algoritmo 43. Secuencias para el tratamiento de listas unidireccionales

La busqueda de (el nombre de) un elemento con un valor entero prefijado n puede expresarse

si existe q **en** *elem(p)* **talque** $q\uparrow.e = n$ **entonces** ...

o, sin utilizar las secuencias, como

```
q ← p;
mientras q ≠ nulo y entonces q↑.e ≠ n haz -- Notese la necesidad de y entonces
  q ← q↑.siguiente;
fin mientras;
-- q = nulo o q↑.e = n
```

Arboles

Un *arbol binario* (o simplemente *árbol*) se define (recursivamente) como

Un arbol es
- o *el arbol vacio (que no consta de ningun elemento)*
- o *un elemento (llamado nodo)*
  - y *dos arboles (subarbol izquierdo y subarbol derecho)*

o, en UBL,

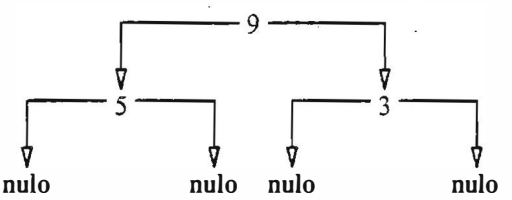
```
tipo arbol es nombre
  tupla
    nodo: entero;
    sub_i,sub_d: arbol;
  fin tupla;
```

De la definicion se deduce (suponiendo que los elementos o nodos son enteros, que **nulo** simboliza el arbol vacio, y que *ARBOL(elemento,arbol1,arbol1)* es el arbol cuyo nodo es *elemento* y sus subarboles *arbol1* y *arbol2*) que son arboles las siguientes construcciones

nulo
*ARBOL(0,***nulo**,**nulo***)*
*ARBOL(-5,ARBOL(0,***nulo**,**nulo***),***nulo***)*
*ARBOL(2,ARBOL(3,***nulo**,*ARBOL(0,***nulo**,**nulo***)),***nulo***)*
... etc.

Los arboles suelen representarse en forma de arbol invertido (de ahi su nombre). Al nodo situado "mas arriba" se le suele llamar *raiz*; y a los subarboles situados "mas abajo" (esto es, aquellos cuyos dos subarboles son el arbol vacio), *hojas*.

*ARBOL(9,ARBOL(5,***nulo**,**nulo***),ARBOL(3,***nulo**,**nulo***))*



Raiz: 9. *Hojas: 5,3.* *Nodos: 9,5,3.*

Figura 87. Representacion de un arbol binario

Las operaciones mas usuales sobre arboles incluyen funciones para hallar el valor de la raiz, y los subarboles de un arbol dado,

```
funcion raiz(a: arbol): entero haz vale a↑.nodo; fin;

funcion sub_i(a: arbol): arbol haz vale a↑.sub_i; fin;

funcion sub_d(a: arbol): arbol haz vale a↑.sub_d; fin;
```

Algoritmo 44. Funciones para el manejo de arboles: En todos los casos se ha supuesto a ≠ **nulo**; el tipo del campo *nodo (entero)* se utiliza unicamente como ejemplo.

acciones para insertar elementos en el arbol segun determinado criterio

```
accion inserta(n: entero; var a: arbol) haz
  -- Inserta un nodo de valor N en el arbol A suponiendo que los valores
  -- de todo subarbol izquierdo son menores que el nodo del que dependen,
  -- y que este es menor que los valores de los nodos de su subarbol derecho.
  -- Si ya existe un nodo con valor N, el efecto es nulo.
  decide
    cuando a = nulo ⇒
      crea a;
      con a↑ haz
        nodo ← n;
        sub_i ← nulo;
        sub_d ← nulo;
      fin con;
    cuando a↑.nodo > n ⇒ inserta n, a↑.sub_i;
    cuando a↑.nodo < n ⇒ inserta n, a↑.sub_d;
    cuando otros ⇒ nada;
  fin decide;
fin inserta;
```

Algoritmo 45. Insercion en un arbol binario

y secuencias para "recorrer" el arbol en algun orden. Los ordenes mas conocidos para recorrer un arbol son los siguientes (su definicion, como la del arbol, es tambien recursiva):

- Recorrer un arbol en *inorden* es

  (1) recorrer el subarbol izquierdo
  (2) pasar por la raiz
  (3) recorrer el subarbol derecho

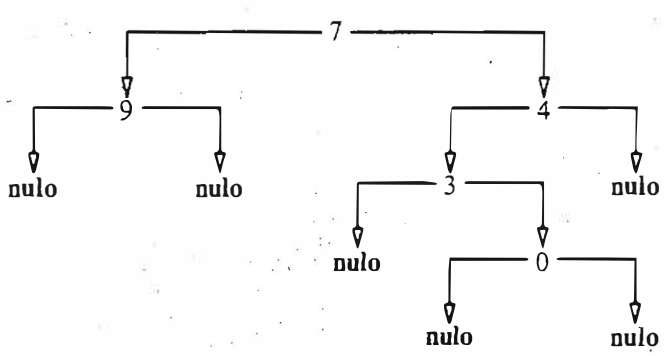
- Recorrer un arbol en *preorden* es

  (1) pasar por la raiz
  (2) recorrer el subarbol izquierdo
  (3) recorrer el subarbol derecho

- Recorrer un arbol en *postorden* es

  (1) recorrer el subarbol izquierdo
  (2) recorrer el subarbol derecho
  (3) pasar por la raiz

La siguiente figura muestra un arbol y los valores obtenidos recorriendolo en preorden, postorden e inorden:



Inorden: 9 7 3 0 4. *Preorden: 7 9 4 3 0.* *Postorden: 9 0 3 4 7.*

Figura 88. Un arbol y sus recorridos

Los recorridos se implementan mediante secuencias, que presentamos aqui en su version recursiva

```
secuencia inorden(a: arbol): entero es
  var n: entero;
haz
  si a ≠ nulo entonces
    para n en inorden(a↑.sub_i) haz produce n; fin;
    produce a↑.nodo;
    para n en inorden(a↑.sub_d) haz produce n; fin;
  fin si;
fin inorden;

secuencia preorden(a: arbol): entero es
  var n: entero;
haz
  si a ≠ nulo entonces
    produce a↑.nodo;
    para n en preorden(a↑.sub_i) haz produce n; fin;
    para n en preorden(a↑.sub_d) haz produce n; fin;
  fin si;
fin preorden;

secuencia postorden(a: arbol): entero es
  var n: entero;
haz
  si a ≠ nulo entonces
    para n en postorden(a↑.sub_i) haz produce n; fin;
    para n en postorden(a↑.sub_d) haz produce n; fin;
    produce a↑.nodo;
  fin si;
fin postorden;
```

Algoritmo 46. Secuencias inorden, preorden y postreorden sobre un arbol

En el proximo capitulo, al hablar de **modulos**, se estudiaran ejemplos practicos de utilizacion de **nombres**.

Ejercicios

1. Diseñar un algoritmo para insertar elementos en la cola de un lista unidireccional [**Indicacion**: mantener una variable auxiliar con el nombre del elemento final].

2. Diseñar un algoritmo para insertar un elemento $q\uparrow$ *antes* de un elemento de nombre *r* en una lista unidireccional [**Indicacion**: utilizar, si es preciso, variables auxiliares].

3. Escribir una **accion** que tome como parametro (por **variable**) la cabeza de una lista unidireccional, invierta esa lista, y modofique el parametro para que siga apuntando a la cabeza de la lista invertida (lo que antes era la cola).

4. Escribir una **secuencia** no recursiva para atravesar un arbol binario en inorden (**Indicaciones**: puede probarse de ampliar la definicion de tipo *arbol* para que incluya el **nombre** del nodo del cual cada nodo "desciende").

Modulos

Los **modulos** son mecanismos sintacticos generales que pueden utilizarse con muy diversos propositos: encapsulacion de entidades logicamente relacionadas, proteccion de la estructura interna de un tipo de datos, control selectivo de la visibilidad de identificadores, etc. Sintacticamente, constan de dos partes, que llamaremos *parte de definicion* y *parte de implementacion*.

```
declaracion_de_modulo = definicion_de_modulo | implementacion_de_modulo

definicion_de_modulo =
  modulo identificador es
    definicion
    {definicion}
  fin [identificador];

definicion = declaracion_de_objeto
  | declaracion_de_tipo
  | cabecera ;
  | definicion_de_modulo

implementacion_de_modulo =
  modulo implementa identificador es
    {declaracion}
  [haz
    instruccion
    {instruccion}]
  fin [identificador];
```

Figura 89. Sintaxis de las declaraciones de **modulo**

La parte de definicion puede contener declaraciones de objetos (constantes o variables), de tipos, *definiciones* de subprogramas (vease "Recursividad" en la pagina 127) y partes de definicion de otros modulos; la parte de implementacion (que puede omitirse si la parte de definicion consta solo de declaraciones de objetos) *debe* contener las implementaciones de los subprogramas, y las partes de implementacion de los modulos, definidos en la parte de definicion, y *puede* contener declaraciones adicionales de objetos, tipos, modulos o subprogramas "internos" (el sentido en el que decimos "internos" quedara claro en seguida), asi como una parte ejecutable, que puede utilizarse, por ejemplo, para inicializar los objetos declarados. Las partes ejecutables de los modulos se ejecutan, en el orden en que estan declarados, inmediatamente despues de la activacion del bloque que los declara.

Las dos partes de un modulo deben hallarse en la misma parte declarativa, la definicion antes de la implementacion, pero no necesariamente juntas (esto es, pueden encontrarse otras declaraciones entre la definicion y la implementacion).

Las reglas de reconocimiento de nombres se alteran al utilizar **modulos**. Los identificadores declarados en la parte de definicion pueden utilizarse si media una declaracion **usa**: en este sentido, los identificadores pueden "salir" de los parentesis sintacticos en los que se definen, cosa que no sucede mas que con **modulos**; los declarados en la implementacion son siempre locales al **modulo**, y no pueden utilizarse desde otra parte.

Las entidades declaradas en un modulo pueden hacerse *visibles* (o sea, conocidas o accesibles) en un bloque mediante la declaracion **usa**:

declaracion_usa = **usa** *identificador* {,*identificador*};

Figura 90. Sintaxis de la declaracion **usa**

Esta declaracion puede utilizarse inmediatamente despues del modulo de definicion (y antes de la implementacion), o en cualquier otro sitio; su efecto es añadir al espacio de identificadores utilizables el de las entidades definidas en el modulo. Las unicas entidades visibles (a traves de la declaracion **usa**) son las de la parte de definicion, mientras que las de la parte de implementacion quedan ocultas, y no pueden utilizarse fuera del modulo. Este mecanismo permite, entre otras cosas, el *control selectivo de la visibilidad* que se habia anunciado:

```
modulo M es
  var i,j: entero;
fin M;

accion A1 es
 -- No puede acceder a I o J a menos que haga usa M
fin A1;

accion A2 es usa M; haz
 -- Puede acceder a I o J directamente
fin A1;
```

En caso de que se esten **usa**ndo varios **modulos** que contengan declaraciones de entidades (no necesariamente distintas) con el mismo identificador, ese identificador pasa a ser *invisible*, escondido o no utilizable.

```
modulo M1 es
  var i,j: entero;
fin M1;

modulo M2 es
  var j,k: entero;
fin M2;

accion A1 es usa M1; haz
  -- puede utilizar I y J (de M1)
fin A1;

accion A2 es usa M2; haz
  -- puede utilizar K y J (de M2)
fin A2;

accion A3 es usa M1, M2; haz
  -- puede utilizar I y K, pero no J (de quien?)
fin A3;
```

Un caso tipico en que los modulos son utiles es al diseñar un "paquete" de subprogramas relacionados con algun tema: por ejemplo, un conjunto de subprogramas matematicos:

```
modulo Funciones_matematicas_elementales es

  const pi = 3.1415926535;
  const e  = 2.7182818284;

  funcion seno(x: real): real;
  funcion coseno(x: real): real;
  funcion senh(x: real): real; -- seno hiperbolico
  funcion cosenh(x: real): real; -- coseno hiperbolico
  funcion exp(x: real): real; -- E elevado a X
  funcion ln(x: real): real; -- logaritmo neperiano
  funcion log(x: real): real; -- logaritmo decimal
  funcion raiz(x: real): real; -- raiz cuadrada

fin Funciones_matematicas_elementales;

modulo implementa Funciones_matematicas_elementales es
  -- aqui se escribirian las implementaciones de
  -- cada uno de los subprogramas definidos mas arriba
fin Funciones_matematicas_elementales;
```

Algoritmo 47. Subprogramas matematicos

Lo interesante es que un programador que *utilice* (puede ser una persona distinta del que lo escribe) el **modulo** anterior solo debe "entender" (conocer el significado de) la parte de definicion, y por tanto puede prescindir de la implementacion; de hecho, si conviene, quien escribio el **modulo** puede cambiar (o corregir, si se revelase erronea) la implementacion de cualquier subprograma (por ejemplo, substituyendola por otra mas eficiente o mas exacta) sin que los que lo **usan** observen cambio alguno. Todo lo cual es idoneo como soporte de metodologias modulares de programacion.

Dos implementaciones de un mismo problema

Como ejemplo presentamos el siguiente programa, que define un **modulo** del que damos dos implementaciones distintas. El problema tratado es sencillo: se trata de averiguar, dado un texto, la frecuencia de aparicion de cada una de las palabras del texto (no se ha tenido en cuenta la presencia de signos de puntuacion, pero el programa puede readaptarse facilmente).

El metodo utilizado consiste en postular la existencia de una tabla extensible (abstracta, no como las **tablas** del lenguaje; como las tablas de los libros, p. ej. una tabla de pesos atomicos) que va registrando las frecuencias de las palabras:

palabra	*frecuencia*
control	*5*
cuenta	*1*
deficiencias	*4*
del	*9*
en	*12*
errores	*3*
graves	*2*
las	*5*
sin	*13*
tener	*1*
y	*1*

Figura 91. Tabla de palabras y sus frecuencias de aparicion

En esta tabla, las palabras estan ordenadas, y van insertandose (en la posicion correcta) las nuevas a medida que se encuentran; inicialmente, esta vacia.

El programa (incompleto, ya que falta la implementacion del **modulo** *frecuencias*) supone que tratamos la tabla mediante las siguientes operaciones:

- *valor(p)*, donde *p* es una palabra, es la frecuencia de aparicion de la palabra, que se supone encontrada al menos una vez;
- *esta(p)* es una **condicion** que nos dice si una palabra nueva aparece por primera vez (**no** *esta(p)*) o no (*esta(p)*) en la tabla;
- *cabenmas* indica si se ha llenado la tabla (suponiendo, lo cual, como se vera, puede ser *falso*) que su capacidad esta acotada;
- *davalor p,n* distingue entre dos casos: si la palabra *p* ya esta en la tabla, cambia su frecuencia, que pasa a valer *n*; si no esta (y *cabenmas*), la mete en la tabla, con frecuencia *n*; por ultimo,
- *valores* es una **secuencia** que produce la lista ordenada de las palabras y sus frecuencias (a ser utilizada al final del proceso).

Se notara que, con esta descripcion, no hacemos suposicion alguna sobre la forma en que se representara la tabla, sino que solo anunciamos cuales son sus propiedades. Esto es sifuciente para entender el siguiente programa:

```
programa analiza_frecuencias es

  tipo clave es tira(10);

  modulo frecuencias es
    funcion valor(const c: clave): entero;
    condicion esta(const c: clave);
    condicion cabenmas;
    accion davalor(const c: clave; i: entero);
    tipo par es tupla c: clave; n: entero; fin;
    secuencia valores: par;
  fin frecuencias;

  -- aqui deberia figurar la parte de implementacion del modulo FRECUENCIAS;
  -- las dos figuras que siguen son versiones correctas de tal implementacion

  usa frecuencias;
  -- permite utilizar las entidades definidas en el modulo Analiza_frecuencias

  var ch: caracter; (* ultimo caracter leido *)

  accion lee_tira(var c: clave) haz
    c ← ""; mientras no fdf y ch = ' ' haz lee ch; fin;
    si no fdf entonces
      mientras ch ≠ ' ' haz c ← c || tira(ch); lee ch; fin;
    fin si;
  fin lee_tira;

  var c: clave; todo_bien: logico; p: par;

haz

  lee ch; todo_bien ← cierto;

  itera
    lee_tira c;
  sal cuando no todo_bien o sino c = "";
    decide
      cuando esta(c) ⇒ davalor c,valor(c)+1; -- incrementa frecuencia
      cuando cabenmas ⇒ davalor c,1; -- crea frecuencia
      cuando otros ⇒ todo_bien ← falso; -- no caben mas
        escribe_linea "Error en la tabla de frecuencias";
    fin decide;
  fin itera;

  si todo_bien entonces
    para p en valores haz escribe_linea "La palabra '",p.c,"' aparece ",p.n," veces."; fin;
  fin si;

fin programa;
```

Algoritmo 48. (Incompleto) Frecuencia de aparicion de palabras en una frase

Distintos modelos o *implementaciones* pueden servir para los objetivos que nos hemos propuesto. La version que sigue utiliza un arbol binario: en cada nodo se encuentra una palabra con su frecuencia asociada; el arbol esta ordenado de modo que la palabra de todo nodo es mayor que toda palabra de su subarbol izquierdo, y mayor que toda palabra de su subarbol derecho (lo cual [vease el Algoritmo 45 en la pagina 141] nos permite obtener la lista ordenada mediante un recorrido en inorden [Cfr. el Algoritmo 46 en la pagina 142]). La condicion *cabenmas* es siempre *cierto*, ya que (aparte de las limitaciones impuestas por la memoria disponible) no hay limite superior al numero de nodos de un arbol implementado mediante **nombres**. La parte ejecutable de la implementacion establece la condicion inicial de tabla vacia (*raiz* ← **nulo**).

```
modulo implementa frecuencias es

  tipo arbol es nombre tupla ant,sig: arbol; cl: clave; n: entero; fin;
  var raiz: arbol;

  funcion valor(const c: clave): entero es
    var act: arbol;
  haz
    si raiz ≠ nulo entonces
      act ← raiz;
      repite
        con act↑ haz
          decide
            cuando cl = c ⇒ vale n;
            cuando cl < c ⇒ act ← sig;
            cuando otros  ⇒ act ← ant;
          fin decide;
        fin con;
      hastaque act = nulo;
    fin si;
    (* Error : clave erronea *)
  fin valor;

  condicion esta(const c: clave) es
    var act: arbol;
  haz
    si raiz ≠ nulo entonces
      act ← raiz;
      repite
        con act↑ haz
          decide
            cuando cl = c ⇒ vale cierto;
            cuando cl < c ⇒ act ← sig;
            cuando otros  ⇒ act ← ant;
          fin decide;
        fin con;
      hastaque act = nulo;
    fin si;
    vale falso;
  fin esta;

  condicion cabenmas haz vale cierto; fin;

  accion davalor(const c: clave; i: entero) es
    var act,antr: arbol; izq: logico;
  haz
    si raiz = nulo entonces
      crea raiz;
      con raiz↑ haz cl ← c; sig ← nulo; ant ← nulo; n ← i; fin;
      acaba;
    sino
      act ← raiz; antr ← raiz;
      repite
        con act↑ haz
          si cl = c entonces n ← i; acaba;
          sino
            antr ← act;
            si cl < c entonces act ← sig; izq ← falso; sino  act ← ant; izq ← cierto; fin;
          fin si;
        fin con;
      hastaque act = nulo;
    fin si;
    crea act;
    si izq entonces antr↑.ant ← act; sino antr↑.sig ← act; fin;
    con act↑ haz cl ← c; n ← i; ant ← nulo; sig ← nulo; fin;
```

```
fin (* esta *);

secuencia valores: par es
  secuencia valores2(a: arbol): par es var p: par; haz
    si a ≠ nulo entonces
      con a↑ haz
        para p en valores2(ant) haz produce p; fin;
        p.c ← cl; p.n ← n; produce p;
        para p en valores2(sig) haz produce p; fin;
      fin con;
    fin si;
  fin valores2;
  var p: par;
haz
  para p en valores2(raiz) haz produce p; fin;
fin valores;

haz
  raiz ← nulo; fin frecuencias;
```

Algoritmo 49. Tabla de frecuencias mediante nombres

Otra version implementa el arbol en forma de **tabla**, substituyendo los **nombres** por (sub)indices de la tabla. El papel que jugaba **nulo** en la anterior figura lo realiza aqui el indice 0; en este caso, *cabenmas* no es siempre *cierto*, sino que depende del numero de palabras encontradas: el espacio para un nuevo nodo se obtiene de la primera posicion *libre* de la tabla (en vez de mediante la operacion *crea*), que puede no existir (*libre* > *100*). En lo demas, el esquema es el mismo.

```
modulo implementa frecuencias es

  var t: tabla[1..100] de tupla ant,sig: [0..100]; cl: clave; n: entero; fin;
  var libre: [1..101];

  funcion valor(const c: clave): entero es
    var act: [0..100];
  haz
    si libre > 1 entonces
      act ← 1;
      repite
        con t[act] haz
          decide
            cuando cl = c ⇒ vale n;
            cuando cl < c ⇒ act ← sig;
            cuando otros ⇒ act ← ant;
          fin decide;
        fin con;
      hastaque act = 0;
    fin si;
    (* Error : clave erronea *)
  fin valor;

  condicion esta(const c: clave) es
    var act: [0..100];
  haz
    si libre > 1 entonces
      act ← 1;
      repite
        con t[act] haz
          decide
            cuando cl = c ⇒ vale cierto;
            cuando cl < c ⇒ act ← sig;
            cuando otros ⇒ act ← ant;
          fin decide;
```

```
      fin con;
    hastaque act = 0;
  fin si;
  vale falso;
fin esta;

condicion cabenmas haz vale libre ≤ 100; fin;

accion davalor(const c: clave; i: entero) es
  var act,antr: [0..100]; izq: logico;
haz
  si libre = 1 entonces
    libre ← 2;
    con t[1] haz cl ← c; sig ← 0; ant ← 0; n ← i; fin;
    acaba;
  sino
    act ← 1; antr ← 1;
    repite
      con t[act] haz
        si cl = c entonces n ← i; acaba;
        sino
          antr ← act;
          si cl < c entonces act ← sig; izq ← falso; sino act ← ant; izq ← cierto; fin;
        fin si;
      fin con;
    hastaque act = 0;
  fin si;
  libre ← libre + 1;
  si izq entonces t[antr].ant ← libre; sino t[antr].sig ← libre; fin;
  con t[libre] haz cl ← c; n ← i; ant ← 0; sig ← 0; fin;
fin (* esta *);

secuencia valores: par es
  secuencia valores2(i: entero): par es
    var p: par;
  haz
    con t[i] haz
      si ant ≠ 0 entonces para p en valores2(ant) haz produce p; fin; fin;
      p.c ← cl; p.n ← n; produce p;
      si sig ≠ 0 entonces para p en valores2(sig) haz produce p; fin; fin;
    fin con;
  fin valores2;

  var p: par;

haz
  si libre > 1 entonces para p en valores2(1) haz produce p; fin; fin si;
fin valores;

haz
  libre ← 1; fin frecuencias;
```

Algoritmo 50. Tabla de frecuencias mediante tablas

Con este programa hemos mostrado que una idea (especificada en forma de un conjunto de operaciones) puede ser realizada o implementada de distintos modos, y que los **modulos** proporcionan el mecanismo apropiado para hacerlo (Incidentalmente, se ha dado ademas una implementacion sin **nombres** de un arbol).

El siguiente ejemplo, algo artificioso, intenta mostrar las posibilidades de proteccion que ofrecen los **modulos**.

Acceso controlado a variables

Supongamos esencial para el correcto funcionamiento de un programa que una variable entera *n* contenga solo valores impares. Ello puede conseguirse de varios modos:

- En primer lugar, puede anadirse una instruccion de invocacion del estilo de *verifica n;* despues de cualquier potencial alteracion de *n*, suponiendo que *verifica* es una accion que controla si efectivamente el valor de *n* es impar. Esto tiene el inconveniente de ser tedioso al obligar a escribir mucho; ademas, puede ser que se olvide algun punto en el que "*n* es impar" debe verificarse.
- Otra alternativa es diseñar subprogramas que realicen las operaciones de inspeccion y modificacion sobre *n*, y evitar sistematicamente cualquier referencia directa a *n*. Esta solucion es mas comoda, pero mantiene el inconveniente de que, si por inadvertencia en algun lugar se modifica directamente *n*, el compilador no lo detecta, con lo cual un solo error puede romper todo el esquema. Esto nos lleva directamente a que
- En un tercer esquema, se desearia utilizar la estructura del segundo, pero prohibir de algun modo el acceso a *n*. Para ello, se utiliza un **modulo**, cuya funcion en este caso es *esconder* la variable *n* de inspecciones o modificaciones no autorizadas.

```
modulo manejo_de_N es
  accion N_pasa_a_valer(m: entero);
  funcion valor_de_N: entero;
fin manejo_de_N;

modulo implementa manejo_de_N es

  var N: entero; -- oculta

  accion N_pasa_a_valer(m: entero) haz
    si impar(m) entonces N ← m;
    sino escribe_linea "Error en el manejo de N";
    fin si;
  fin N_pasa_a_valer;

  funcion valor_de_N: entero haz vale N; fin;

haz
  n ← 1;
fin manejo_de_N;
```

Algoritmo 51. Acceso controlado a una variable

Pilas

Una de las aplicaciones principales de las listas definidas en el capitulo anterior se encuentra al tratar de implementar *pilas*. Una pila puede concebirse como un tubo cerrado por un extremo (p. ej., un tubo de pastillas) donde se pueden "meter" y "sacar" datos; los datos pueden acumularse en la pila, pero, al sacarlos, siempre salen en el orden inverso a aquel en que se han metido. Es suficiente el siguiente conjunto de operaciones para manejar una pila:

- *Mete e* mete el elemento *e* en la pila;
- *Saca* saca un elemento de la pila (y lo "destruye");
- *Superior* da el valor del elemento visible o "mas externo" de la pila; por ultimo,
- *Vacia* y *llena* son condiciones que indican el estado de la pila

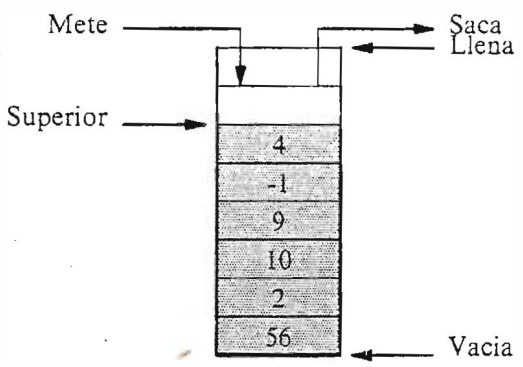


Figura 92. Una pila de enteros: El sombreado representa la parte utilizada.

Damos una definicion y dos implementaciones de las pilas; entre estas ultimas, una es no acotada (tubo con capacidad infinita o infinitamente largo) y se realiza con **nombres** y otra es acotada y utiliza **tablas** (comparese con las implementaciones de la tabla de frecuencias presentadas mas arriba).

```
modulo pila es
  accion mete(e: entero);
  accion saca;
  funcion superior: entero;
  condicion vacia;
  condicion llena;
fin pila;
```

Algoritmo 52. Definicion de una pila de enteros

```
modulo implementa pila es

  tipo pila es nombre tupla e: entero; p: pila; fin;
  var p: pila;

  accion mete(e: entero) es var q: pila; haz
    crea q; q↑.e ← e; q↑.p ← p; p ← q;
  fin mete;

  accion saca haz
    si vacia entonces escribe_linea "Error"; sino p ← p↑.p; fin;
  fin saca;

  funcion superior: entero;
    si vacia entonces escribe_linea "Error"; sino vale p↑.e; fin;
  fin superior;

  condicion vacia haz vale p = nulo; fin;
  condicion llena haz vale falso; fin;

haz
  p ← nulo;
fin pila;
```

Algoritmo 53. Implementacion de una pila de enteros mediante nombres

```
modulo implementa pila es

  var p: tabla [1..100] de entero; i: entero;

  accion mete(e: entero) es var q: pila; haz
    si llena entonces escribe_linea "Error"; sino i ← i + 1; p[i] ← e; fin;
  fin mete;

  accion saca haz
    si vacia entonces escribe_linea "Error"; sino i ← i - 1; fin;
  fin saca;

  funcion superior: entero;
    si vacia entonces escribe_linea "Error"; sino vale p[i]; fin;
  fin superior;

  condicion vacia haz vale i = 0; fin;
  condicion llena haz vale i = 100; fin;

haz
  i ← 0;
fin pila;
```

Algoritmo 54. Implementacion de una pila de enteros mediante tablas

Apendice 1: Pascal ISO y Pascal/VS a partir de UBL

Introduccion

El lenguaje Pascal fue diseñado por Niklaus Wirth hacia 1968; en 1970 aparecio una version revisada. Pascal fue concebido como instrumento para la enseñanza sistematica de la programacion, y de modo que fuese facilmente implementable en la mayoria de ordenadores. Pese a no contar con soportes oficiales ni comerciales, debido a su simplicidad, elegancia conceptual y riqueza expresiva fue extendiendose entre los programadores de todo el mundo. Ha sido uno de los puntos de partida de numerosas investigaciones sobre lenguajes.

Tal como esta definido en [JWi 78], el lenguaje Pascal contiene algunas inconsistencias y puntos poco claros: la Asociacion Internacional de Standards (ISO) ha elaborado una definicion del lenguaje que intenta corregir esos errores, y que ha pasado a conocerse como Pascal Standard; por su parte cada fabricante, ademas de ajustar mas o menos su compilador a las exigencias del Standard, suele incorporar extensiones para facilitar el acceso a las peculiaridades del sistema operativo, o como soluciones en las areas donde el diseño de Pascal lo convierte en un lenguaje demasiado pobre. La version proporcionada por IBM para sus ordenadores tipo IBM/370 con sistemas operativos VM/CMS y MVS se llama Pascal/VS.

En el resto del apendice describiremos el lenguaje Pascal a partir de UBL. Para ello, y despues de tratar algunas cuestiones de base en que los dos lenguajes difieren, describiremos cada una de las construcciones linguisticas que proporciona Pascal, comentando en cada caso las similitudes y diferencias respecto de UBL. En el caso de construcciones existentes en UBL pero no en Pascal (como es el caso de las secuencias o los modulos), se describiran formulaciones alternativas compatibles con ese lenguaje. Se explicara conjuntamente Pascal Standard y Pascal/VS; dado que el segundo es una extension del primero, las descripciones se aplicaran a los dos lenguajes, indicando lo contrario explicitamente cada vez que sea necesario.

Nota: la siguiente explicacion es introductoria, y *no* suple el estudio detallado de la bibliografia recomendada.

Bibliografia:

[JWi 78] K. Jensen, N. Wirth: *Pascal Users Manual and Report*. Springer-Verlag, New York, 1978.

Es el manual en el que se encuentra la definicion original de Pascal. Mucho mas legible que la version ISO.

[Wi 73] N. Wirth: *Systematic Programming: An Introduction*. Prentice-Hall, Englewood Cliffs, 1973

Introduce la programacion de un modo riguroso basandose en enunciados logicos. Hay version castellana (*Introduccion a la programacion sistematica*) editada por El Ateneo, Buenos Aires, 1982.

[Wi 76] N. Wirth: *Algorithms + Data Structures = Programs*. Prentice-Hall, Englewood Cliffs, 1976

Abarca en profundidad temas como ordenacion de tablas, recursividad, e introduccion a la escritura de compiladores. Hay una edicion en Castellano (*Algoritmos + Estructuras de Datos = Programas*) en Ediciones del Castillo, Madrid, 1984.

[IBM 81] IBM: *Pascal/VS Language Reference Manual*. SH20-6168-1

[IBM 83] IBM: *Pascal/VS Programmer's Guide*. SH20-6162-1

Son los manuales de la version de Pascal para IBM.

Un primer ejemplo

La estructura de Pascal es muy similar a la de UBL: los identificadores reservados y predefinidos estan en ingles, pero un programa sencillo en Pascal (sabiendo un poco de ingles) es casi inmediatamente comprensible.

——— *UBL:* ———

```
programa Multiplicacion_de_Gitanos es
  var m,n,p: entero;
haz
  lee m,n;
  si impar(n) entonces p ← m; sino p ← 0; fin;
  repite
    m ← 2 * m;
    n ← n div 2;
    si impar(n) entonces p ← p + m; fin;
  hastaque n = 1;
  escribe p;
fin programa;
```

——— *Pascal:* ———

```
program MultiplicacionDeGitanos (input,output);
  var m,n,p: integer;
begin
  read(m,n);
  if odd(n) then p := m else p := 0;
  repeat
    m := 2 * m;
    n := n div 2;
    if odd(n) then p := p + m
  until n = 1;
  write(p)
end.
```

Figura 93. El programa de la multiplicacion de Gitanos en UBL y en Pascal: La estructura del programa es practicamente identica. La asignacion en Pascal se escribe ":="; se notara la ausencia de el **end** (**fin**) despues del **if**, y la de algunos puntos y coma.

Instrucciones

Uso del punto y coma

El punto y coma (";") se utiliza en UBL como *terminador* de instrucciones, y en Pascal como *separador* de estas. De otro modo: en UBL, cada instruccion lleva su propio punto y coma, mientras que en Pascal solo debe escribirse un punto y coma cuando sea necesario para separar dos instrucciones.

Por ejemplo, las instrucciones UBL

```
a ← b; c ← d;
```

se escriben en Pascal como

```
a := b; c := d
```

y el hecho de que la ultima asignacion lleve o no punto y coma dependera de que siga a ese fragmento de programa. Por otra parte, la instruccion UBL

si ... entonces *p* sino *q*. fin.

se escribe en Pascal

if *a* < *b* then *p* else *q*

omitiendo (ademas del fin que cierra el si, como se vera mas adelante) el punto y coma despues de la invocacion *p*, ya que else (sino) no es una instruccion, sino un separador, y por tanto no es necesario ningun punto y coma [De hecho, la presencia de un punto y coma despues de *p* haria incorrecto el programa].

Desde el punto de vista teorico, es interesante la utilizacion del punto y coma que hace Pascal: el simbolo ";" se concibe como un operador que puede leerse "y despues haz", y se toma

p; *q*

como una forma de decir "ejecuta la instruccion *p y despues* (esto es, una vez terminada la ejecucion de *p*) ejecuta la instruccion *q*", subrayando asi el caracter secuencial del lenguaje. Sin embargo, desde el punto de vista del que debe escribir un programa, tal uso obliga a hacer distinciones que no siempre son interesantes, y complica extraordinariamente la modificacion de programas ya existentes.

Falta de parentizacion en la sintaxis

Otra caracteristica de Pascal que lo hace diferente de UBL es la forma que tiene de definir su sintaxis. Por ejemplo, la instruccion **if** (**si**) se define como

instruccion_if = **if** *condicion* **then** *instruccion* [**else** *instruccion*]

Por un lado, no hay ningun "**end if**" o construccion equivalente que marque el final del **if**; por otro, despues de **then** (o de **else**) puede ir *una instruccion, pero no varias*. Por supuesto, existe una forma de englobar varias instrucciones en una sola:

begin
instruccion$_1$;
instruccion$_2$;
...
instruccion$_n$
end

es una *instruccion compuesta*, y puede figurar como instruccion despues de **then** o **else**. Ello obliga a escribir diferentemente una instruccion **if** que condicione una sola instruccion y una que condicione varias: si en

if *c* **then** *p*

se desea cambiar *p* por *p; q*, debe escribirse

if *c* **then begin** *p; q* **end**

y no

if *c* **then** *p; q*

que se interpretaria como la construccion UBL

si *c* **entonces** *p;* **fin**; *q;*

Esto es causa de frecuentes errores en la modificacion de programas escritos en Pascal.

Diferencias lexicas

- El caracter de subrayado ("_") no forma parte (al igual que los caracteres 'ñ' y 'Ñ' del alfabeto basico de Pascal; asi, no es posible construir identificadores que contengan el simbolo "_", por

lo que se ha de recurrir a mezclas de mayusculas y minusculas para distinguir, si se desea, las palabras que lo forman, como en

PasaPagina
NoDeLineas

Pascal/VS no tiene esta limitacion, y ademas permite usar el caracter "S" como si fuese una letra. De hecho, el subrayado puede utilizarse a final de identificador o varias veces seguidas.

```
A_Y_B
A__Y__B (* distinto del anterior *)
A_______$_$$$__
```

cosa no permitida en UBL.

- Las tiras constantes o literales de tira se encierran en Pascal entre apostrofes (') en vez de entre comillas ("): ademas, como se vera, son valores de tipos distintos, ya que Pascal Standard no trata tiras ajustables de caracteres (aunque si lo hace Pascal/VS).

```
'Esto es una tira de caracteres en Pascal Standard'
```

- Los comentarios en Pascal se encierran entre "(*" y "*)"; los simbolos "{" y "}" se consideran equivalentes a los primeros

```
(* esto es un comentario *)
{ igual que esto }
{ y esto: aunque no se aconseja *)
(* el uso de las dos ultimas formas }
(* Naturalmente, las llaves no se
utilizan para encerrar conjuntos *)
```

No hay comentarios hasta fin de linea (el simbolo "--" no existe en Pascal). Pascal/VS define una segunda forma de comentario (encerrado entre "/*" y "*/") y permite comentarios (de un tipo) dentro de comentarios (de otro tipo).

- En Pascal, los conjuntos se encierran entre corchetes ([,]) y no entre llaves ({,}).

Pasamos a describir las diferentes instrucciones utilizables en Pascal.

Instrucciones Simples

---

instruccion_simple = instruccion_vacia
| instruccion_de_asignacion
| instruccion_de_invocacion
| instruccion_goto

Figura 94. Instrucciones simples en Pascal

---

instruccion_vacia =

Figura 95. Sintaxis de la instruccion vacia en Pascal

La *instruccion vacia* no contiene ningun simbolo, y forma parte del lenguaje con objeto de permitir relajar en algunos casos las reglas sobre el uso del punto y coma:

begin *p; q* **end**

y

begin *p; q;* **end**

tienen el mismo efecto, ya que puede considerarse que, en el segundo caso, despues de c se ejecuta una instruccion vacia.

El efecto de la instruccion vacia es nulo; equivale a la instruccion **nada**; de UBL.

---

instruccion_de_asignacion = *variable* := *expresion*
| *identificador_de_funcion* := *expresion*

Figura 96. Sintaxis de la instruccion de asignacion en Pascal

---

La *instruccion de asignacion* se utiliza en Pascal de dos modos distintos: en su primera forma, tiene un efecto igual al de su correspondiente en UBL, de la que solo se diferencia por la ausencia del punto y coma final y por utilizar ":=" en vez de "←" como operador de asignacion; en su segunda forma, establece un valor (quiza provisional) al evaluar una funcion, y se estudiara mas adelante. En los dos casos se aplican las mismas reglas de compatibilidad de tipos entre las dos partes de la asignacion que en UBL.

Son ejemplos de asignacion en Pascal

```
a := b
c := d + e; d := c - e; e := c - d
A[i] := A[j]
```

---

instruccion_de_invocacion = *identificador_de_procedure* "(" *lista_de_parametros_actuales* ")"

Figura 97. Sintaxis de la instruccion de invocacion en Pascal: Pascal llama **procedures** a las **acciones**, *parametros formales* a los parametros, y *parametros actuales* a los argumentos.

La instruccion de invocacion en Pascal se diferencia de la de UBL por encerrar sus argumentos entre parentesis; por lo demas, tiene sus mismas propiedades y efectos (a excepcion de la posibilidad de pasar parametros constantes por nombre, que solo ofrece Pascal/VS, pero no Pascal Standard, como se vera al estudiar las declaraciones de suprograma).

Son ejemplos de instruccion de invocacion

```
P(2,3,4)
Hanoi(A,B,C,n-1)
WriteLn(n:6, k:4)
```

---

instruccion_goto = **goto** *etiqueta*

Figura 98. Sintaxis de la instruccion &goto. en Pascal

La instruccion **goto** (literalmente: "ve a") no tiene equivalente en UBL, y merece ser explicada con algo de detalle. La idea es definir "puntos" del programa identificandolos mediante nombres o "etiquetas", y suponer que el interprete obedece instrucciones **goto** que hagan referencia a esos puntos con el efecto de continuar la ejecucion en el lugar identificado:

```
        instruccion1;
        instruccion2;
AQUI:   instruccion3;
        instruccion4;
        instruccion5;
        goto AQUI
(* Al ejecutarse goto AQUI, el programa "salta" y pasa a ejecutar la "instruccion3" *)
```

La instruccion **goto**, que puede considerarse "de bajo nivel" en contraposicion a las demas instrucciones, que serian "de alto nivel", puede utilizarse para implementar la mayoria de las demas instrucciones no simples:

mientras *C* haz
  I
fin mientras;
	A: si no *C* entonces goto *B*; fin;
	  I
	  goto *A*;
	B: nada;

repite *I* hastaque *C*;	*A*: *I* si no *C* entonces goto *A*; fin;

itera
  I1
sal cuando *C*;
  I2
fin itera;
	A:
	  I1
	si *C* entonces goto *B*; fin;
	  I2
	goto *A*;
	B: nada;

Figura 99. ***Descomposicion de algunas instrucciones utilizando &goto.:*** en esta figura, se ha supuesto que UBL esta extendido mediante la instruccion **goto**; las etiquetas que aparecen se suponen declaradas y distintas de los demas identificadores del programa.

Las etiquetas (que en Pascal Standard han de ser enteros sin signo entre 1 y 9999, y en Pascal/VS pueden ser ademas identificadores) deben declararse mediante la declaracion **label**, que se estudiara. El lenguaje prohibe "saltar dentro" de una instruccion

```
   begin
     I1;
A:   I2;
     I3
   end;
   goto A (* incorrecto *)
```

pero permite (aunque bastantes versiones de Pascal no ofrecen esta posibilidad) hacer un **goto** a una etiqueta que se encuentre en el cuerpo de un subprograma que englobe al que hace el **goto**, con lo que proporciona un metodo comodo para tratar situaciones terminales (es decir, aquellas en las que, en un punto del (sub)programa, se decide terminar con la ejecucion de este sin preocuparse de que se estaba ejecutando). El caso mas frecuente puede explicarse con el siguiente esquema

```
program P;
 label 9999;
 ...
 procedure PP;
  ...
  goto 9999;
  ...
 end; (* PP *)
 ...
begin
 ...
9999:
end
```

Figura 100. ***Esquema de uso de &goto. para tratar situaciones terminales:*** Si en algun subprograma (por ejemplo, *PP*) se verifica alguna condicion que hace imposible la continuacion de la ejecucion del programa, se ejecuta **goto** 9999, que "salta" al final del programa principal, y lo termina.

Esta es el unico caso en el que recomendamos el uso de la instruccion **goto**; en los demas casos, es casi siempre posible escribir el programa sin utilizarla.

Instrucciones estructuradas

instruccion_estructurada = instruccion_compuesta
instruccion_condicional
instruccion_repetitiva
instruccion_with

secuencia_de_instrucciones = instruccion { ; instruccion }

Figura 101. Sintaxis general de las instrucciones estructuradas en Pascal

El no terminal *secuencia_de_instrucciones* se introduce como abreviatura para las definiciones que siguen: su efecto es la ejecucion secuencial de las instrucciones que lo componen, excepto si alguna de ellas es **goto**.

instruccion_compuesta = **begin** *secuencia_de_instrucciones* **end**

Figura 102. Sintaxis de la instruccion compuesta en Pascal

La instruccion compuesta es el modo de agrupar varias instrucciones, considerandolas en su totalidad como una sola: "**begin**" y "**end**" pueden considerarse como formas de "parentesis de instrucciones", y se utilizan para encerrar varias cuando la sintaxis requiere solo una.

if *C* **then begin** *p; q* **end**
else begin *r; s; t* **end**

No tiene correspondencia en UBL, ya que en este lenguaje siempre es posible escribir mas de una instruccion donde solo hay una, lo cual hace innecesario el uso de tales parentesis.

instruccion_condicional = *instruccion_if* | *instruccion_case*

Figura 103. Sintaxis general de las instrucciones condicionales en Pascal

instruccion_if = **if** *condicion* **then** *instruccion* [**else** *instruccion*]

Figura 104. Sintaxis de la instruccion **if** *en Pascal*

La instruccion **if** (**si**) tiene el mismo efecto que la instruccion **si** en UBL. Notese que no hay un correspondiente **end if**, y que si alguna de las instrucciones es compuesta debe ir encerrada entre **begin** y **end**. Es importante tambien observar que antes de **else** no puede ir un punto y coma.

if *p(x)* **then** *trata(x)*
else begin *x := x + 3; trata(x)* **end**

Pascal no tiene ninguna instruccion analoga a la **decide** de UBL, pero puede conseguirse el mismo efecto encadenando varias instrucciones **if**:

```
if c1 then i1
else if c2 then i2
  ...
else if cn then in
else io

-- equivale en UBL a

decide
  cuando c1 => i1
  cuando c2 => i2
  ...
  cuando cn => in
  cuando otros => io
fin decide;
```

Figura 105. Modo de obtener el efecto de una instruccion **decide** *mediante varias* **if**

```
instruccion_case =
  case expresion of
    lista_de_constantes_case : instruccion {;
    lista_de_constantes_case : instruccion } [;]
  end

lista_de_constantes_case = constante {, constante}
```

Figura 106. Sintaxis de la instruccion **case** *en Pascal*

La instruccion **case** corresponde a la **segun** de UBL; la *expresion* esta sometida a las mismas restricciones en los dos lenguajes (esto es, debe ser de tipo *entero, caracter, logico*, enumerado o subrango). Pascal Standard no permite especificar rangos ni expresiones estaticas en la lista de constantes; tampoco permite ninguna construccion correspondiente a la "**cuando otros**" de UBL (lo cual fuerza a escribir todos los casos posibles, aunque sea asociados a una instruccion vacia). Estas limitaciones no se aplican a Pascal/VS, que permite rangos, expresiones estaticas, y precedida de "**otherwise**", tratamiento de **otros** casos. Notese que los distintos valores, rangos y constantes se separan por comas y no por barras, que no hay nada que corresponda al simbolo **cuando** y que la flecha ("=>") se substituye por ":". Notese tambien que despues de cada lista de constantes se requiere una sola instruccion, lo que obliga a utilizar **begin** ... **end**si se quiere escribir mas de una. El siguiente ejemplo solo es valido en Pascal/VS:

```
case dia_de_la_semana of
  lunes..miercoles,viernes:  trata_laborable;
  succ(sabado):  { instruccion vacia }
  otherwise trata_resto
end (* case *)
```

```
instruccion_repetitiva = instruccion_repeat | instruccion_while | instruccion_for
```

Figura 107. Sintaxis de las instrucciones repetitivas en Pascal

```
instruccion_repeat =
  repeat
    secuencia_de_instrucciones
  until condicion
```

Figura 108. Sintaxis de la instruccion **repeat** *en Pascal*

La instruccion **repeat** es identica a la **repite** de UBL; todo lo explicado sobre esta se aplica a aquella. Observese que es el unico caso en que una instruccion distinta de la compuesta tiene "parentesis" de abrir y de cerrar.

instruccion_while = **while** *condicion* **do** *instruccion*

Figura 109. Sintaxis de la instruccion **while** *en Pascal*

La instruccion **while** corresponde a la **mientras** de UBL. Se utilizara **begin** ... **end** cuando sea necesario para que **while** afecte a una sola instruccion (compuesta).

instruccion_for =
for *variable_de_control* := *valor_inicial* (**to** | **downto**) *valor_final* **do** *instruccion*

Figura 110. Sintaxis de la instruccion **for** *en Pascal*

Pascal no incluye el concepto de **secuencia**, aunque podria decirse que proporciona dos secuencias predefinidas:

for *v* := *vi* **to** *vf* **do** *I*

equivale a lo que en UBL se escribiria

para *v* **en** *asc(vi.vf)* **haz** *I* **fin**;

Similarmente,

for *v* := *vi* **downto** *vf* **do** *I*

equivale a lo que en UBL se escribiria

para *v* **en** *desc(vi.vf)* **haz** *I* **fin**;

Por otra parte, se exige que la *variable_de_control* este declarada en el mismo cuerpo en que se utiliza la instruccion, y que, dentro de la *instruccion*, no se altere (directa o indirectamente) su valor.

instruccion_with = **with** *lista_de_variables_record* **do** *instruccion*

lista_de_variables_record = *variable_record* {, *variable_record* }

Figura 111. Sintaxis de la instruccion **with** *en Pascal:* "**record**" es el nombre dado en Pascal a las **tuplas**.

La instruccion **with** de Pascal corresponde a la **con** de UBL, con el mismo efecto.

Declaraciones

Orden de las declaraciones en Pascal ISO

Aunque Pascal/VS no lo hace, Pascal Standard impone una limitacion fuerte sobre el orden en que se realizan las declaraciones en un cuerpo. Este orden esta prefijado, y es el siguiente:

declaraciones =
[*declaracion_de_etiquetas*]
[*declaracion_de_constantes*]
[*declaracion_de_tipos*]
[*declaracion_de_variables*]
{*declaracion_de_procedures* | *declaracion_de_funciones*}

Figura 112. Orden de las declaraciones en Pascal Standard

Esto tiene algunos inconvenientes: por ejemplo, no es posible definir un tipo enumerado, y, mas adelante en la misma parte declarativa, definir una constante de ese tipo, ya que la declaracion de constantes debe escribirse antes de la de tipos.

Por otra parte, al estar juntas todas las declaraciones de constantes, de variables, etc., solo se escribe una vez la palabra **const** (**var**, etc) para cada grupo de declaraciones.

Declaracion de etiquetas

declaracion_de_etiquetas = **label** *etiqueta* {,*etiqueta*};

etiqueta = *entero_sin_signo*

Figura 113. Sintaxis de la declaracion de etiquetas (label) en Pascal

Las etiquetas han de ser enteros sin signo comprendidos entre 1 y 9999 (ambos inclusive); Pascal/VS permite tambien identificadores como etiquetas. Todas las etiquetas han de declararse.

Declaracion de constantes

declaracion_de_constantes = **const** *definicion_de_constante*; { *definicion_de_constante*;}

definicion_de_constante = *identificador* "=" *constante*

constante = [*signo*] (*numero_sin_signo* | *identificador_de_constante*) | *tira_de_caracteres*

Figura 114. Sintaxis de las declaraciones de constantes (const) en Pascal

Pascal Standard no permite expresiones constantes en la definicion de constantes; Pascal/VS, si.

Declaracion de tipos

declaracion_de_tipos = **type** *identificador* = *tipo*; {*identificador* = *tipo*}

tipo = *identificador_de_tipo* | *tipo_ordinal* | *tipo_estructurado* | *tipo_pointer*

Figura 115. Sintaxis de las declaraciones de tipo (type) en Pascal

El mecanismo de tipos es similar en Pascal y en UBL, aparte de algunas carencias de Pascal (no hay enumerados no ordenados o ciclicos, ni aplicaciones) y algunas diferencias menores (los tipos **nombre**, llamador *pointer*s en Pascal, siguen un metodo distinto de declaracion).

Tipos ordinales

tipo_ordinal = *identificador_de_tipo_ordinal* | *tipo_enumerado* | *tipo_subrango*

tipo_enumerado = "(" *identificador* {, *identificador*} ")"

tipo_subrango = *constante* .. *constante*

Figura 116. Sintaxis de los tipos ordinales en Pascal

No hay tipos enumerados no ordenados ni ciclicos en Pascal (esto no es un inconveniente grave: para tipos no ordenados, basta con utilizar los ordenados y preocuparse de no utilizar comparaciones ni operaciones de *succ*esor ni *pred*ecesor [aunque el compilador no avisa si se hace]; para tipos ciclicos, hay que tratar "a mano" el paso al *succ*esor o *pred*ecesor si se preve que puede haber conflicto [en Pascal, el *succ*esor del ultimo elemento, y el *pred*ecesor del primero provocan siempre error al calcularlos]); en cuanto a los subrangos, no van encerrados entre corchetes, y

ademas, en Pascal Standard, sus limites han de ser constantes (y no, como en UBL, expresiones constantes). Pascal VS reconoce las siguientes formas:

constante .. expresion_constante

o

range *expresion_constante .. expresion_constante*

Tipos predefinidos

Pascal predefine los mismos tipos que UBL: *integer (entero), real (real), char (caracter)* y *boolean (logico)*.

Tipos estructurados

tipo_estructurado = tipo_array | tipo_record | tipo_set | tipo_file

Figura 117. Sintaxis de los tipos estructurados en Pascal

En Pascal se habla de *tipos estructurados* para referirse colectivamente a los **arrays** (las **tablas** de UBL; Pascal no distingue entre **tabla** y **aplicacion**), **records** (**tuplas**), **sets** (**conjuntos**) y **files** (**filas**). El significado es practicamente el mismo; lo unico que cambia en algunos casos es la sintaxis (aparte de la traduccion al ingles).

Tipos array

tipo_array = **array** *"["tipo_indice {,tipo_indice}"]"* **of** *tipo_elemento*

tipo_elemento = tipo

tipo_indice = tipo ordinal

Figura 118. Sintaxis de los tipos* array *en Pascal

Un **array** es una **tabla** (o **aplicacion**) de UBL. Se notara que los corchetes forman parte de la sintaxis del tipo **array**, a diferencia de en UBL, que atribuia los corchetes, de existir, al subrango que componia el tipo indice. Para casos sencillos, la apariencia es similar:

array *[1..10]* **of** *integer*
tabla *[1..10]* **de** *entero*

pero si hay varios subindices, o si el tipo indice no es un subrango explicito, la escritura difiere:

array *[1..10,1..10]* **of** *integer*
tabla *[1..10], [1..10]* **de** *entero*

array *[boolean]* **of** *char*
tabla *logico* **de** *caracter*

Por lo demas, no hay diferencias.

Tipos record

```
tipo_record =
  record
    lista_de_campos
  end

lista_de_campos = [ (parte_fija [; parte_variante] | parte_variante)[;]]

parte_fija = seccion {; seccion}

seccion = identificador {,identificador}: tipo

parte_variante =
  case selector_de_variante of
    variante {;
    variante}

selector_de_variante = [campo_discriminante: ] tipo_del_discriminante

campo_discriminante = identificador

tipo_del_discriminante = identificador_de_tipo_ordinal

variante = lista_de_constantes_case : "(" lista_de_campos ")"

lista_de_constantes_case = constante {, constante}
```

Figura 119. Sintaxis de los tipos* record *en Pascal

Los **record**s de Pascal corresponden a las **tupla**s de UBL; pueden describirse **record**s sin variantes o con variantes (**case**). Se notara que la(s) parte(s) variante(s) empiezan por **case**, pero no hay ningun correspondiente **end** (**case**): el **end** del **record** realiza esa funcion. Igual que en la instruccion **case**, Pascal Standard (pero no Pascal/VS) solo permite constantes como selectores de variantes. Las variantes (que pueden ser vacias) se encierran entre parentesis, y deben (excepto la ultima, en la que es opcional) ir seguidas de un punto y coma.

```
type representacion = (cartesianas, polares);

type complejo =
  record
    modulo: real;
    case rep: representacion of
      cartesianas: (x,y: real);
      polares: (r,theta: real)
  end;
```

Pascal permite omitir el campo discriminante, escribiendo solo su tipo

```
record
  case boolean of
    true: (i: integer);
    false: (r: real)
end;
```

en ese caso, el tipo del (inexistente) discriminante solo sirve para distinguir (logicamente) las distintas variantes.

Tipos set

```
tipo_set = set of tipo_ordinal
```

Figura 120. Sintaxis de los tipos* set *en Pascal

Los **set**s de Pascal son totalmente equivalentes a los **conjunto**s de UBL.

tipo_file = file of *tipo*

Figura 121. Sintaxis de los tipos file *en Pascal*

Los files de Pascal corresponden a las filas de UBL; la mayoria de operaciones coinciden, con algunas restricciones y diferencias:

Pascal	*UBL*
f↑	*f↑*
get(F)	*obten f;*
put(F)	*pon f;*
read(f,x)	*lee f,x;*
write(f,x)	*escribe f,x;*
reset(f)	*conecta f,"",lectura;*
rewrite(f)	*conecta f,"",escritura;*
eof(f)	*fdf(f)*

Figura 122. ***Relacion entre los subprogramas de entrada y salida en Pascal y UBL:*** La presencia de una tira vacia en la instruccion *conecta* indica que la asociacion entre la fila y el fichero no puede especificarse en las instrucciones *reset* o *rewrite*, y por tanto se realiza exteriormente al programa. La mayoria de compiladores (entre ellos, el de Pascal/VS) admiten un segundo parametro en estos procedures para señalar la ubicacion.

El tipo *text*

en Pascal equivale al tipo *texto* de UBL; es valida la siguiente tabla (ademas de la anterior):

Pascal	*UBL*
eol(f)	*fdl(f)*
writeln(f,...)	*escribe_linea f,...*
readln(f,...)	*lee_linea f,...*

Figura 123. ***Relacion entre el tipo text de Pascal y el tipo texto de UBL:*** Los formatos de Pascal funcionan del mismo modo que los de UBL, con las siguientes restricciones: 1) No se aceptan formatos en lectura, y 2) no hay lectura ni escritura de objetos o valores de tipo enumerado, excepto para le escritura del tipo *boolean*, que si se acepta; y la siguiente diferencia: asi como en UBL, de no haber formato, se utiliza siempre el expacio minimo necesario, en Pascal se supone un formato por defecto dependiente de version (esto significa que, para obtener los mismos resultados en Pascal que en UBL, es necesario escribir siempre ":1" en cada expresion de escritura sin formato.

El tipo estructurado space en Pascal/VS

Pascal/VS define un modo adicional de estructurar los datos.

tipo_space = space *"["expresion_entera"]"* of *tipo_base*

Figura 124. Sintaxis del tipo space *en Pascal/VS*

La *expresion_entera* indica la cantidad de memoria en bytes que ocupara un objeto de tipo **space**. Tal objeto puede indexarse como si se tratase de una tabla, y *en cada indice* se encuentra (potencialmente) un objeto del tipo base.

No debe confundirse una tabla

var *T*: array *[1..10]* of *integer;*

que en un IBM 370 ocupa 40 bytes, con un

var *T*: **space** *[40]* **of** *integer;*

que, aunque ocupa la misma memoria y tambien contiene enteros, *permite tratar mas enteros* (a veces, rampantes): p. ej., *T[11]* o *T[23]* no tienen equivalente en el **array**.

Los tipos **space** se utilizan para implementar mecanismos de memoria dinamica y para tratar bloques COMMON de FORTRAN (definiendo un espacio como **ref** o **def**, tal como se estudiara mas abajo).

Tipos pointer

tipo_pointer = ↑ *tipo*

Figura 125. Sintaxis de los tipos pointer en Pascal

Los tipos pointer de Pascal equivalen a los tipos **nombre** de UBL, con las siguientes diferencias:

- No hay nada similar a las declaraciones incompletas de tipo para tipos mutuamente recursivos; puede hacerse

  type *T* = ↑ *S;*

  antes de que *S* se haya definido (este es el unico caso en que puede violarse la regla de "declaracion antes de uso".

- Pueden declararse objetos o campos directamente como "↑*T*".

Declaracion de variables

declaracion_de_variables =
var
lista_de_identificadores : tipo;
{lista_de_identificadores : tipo;}

lista_de_identificadores = *identificador* {,*identificador*}

Figura 126. Sintaxis de la declaracion de variables (var) en Pascal

Las declaraciones de variable en Pascal son similares a las de UBL.

Declaraciones de subprogramas

```
declaracion_de_procedure = cabecera_de_procedure ; directiva | cabecera_de_procedure ; bloque

declaracion_de_funcion = cabecera_de_funcion ; directiva | cabecera_de_funcion ; bloque

bloque = declaraciones cuerpo

cuerpo = begin {instruccion} end ;

cabecera_de_procedure = procedure identificador ["(" lista_de_parametros ")"]

cabecera_de_funcion = function identificador ["(" lista_de_paranetros ")"] : identificador_de_tipo

lista_de_parametros = parametros {, parametros}

parametros = parametros_por_copia
  | parametros_por_variable
  | parametro_procedure
  | parametro_function

parametros_por_copia = lista_de_identificadores : identificador_de_tipo

parametros_por_variable = var lista_de_identificadores : identificador_de_tipo

parametro_procedure = cabecera_de_procedure

parametro_function = cabecera_de_funcion

lista_de_identificadores = identificador {,identificador}
```

Figura 127. Sintaxis de las declaraciones de subprogramas en Pascal

Pascal solo permite dos tipos de subprograma: **procedures** (**acciones** en UBL) y **functions** (**funciones** en UBL). La ausencia de las **condiciones** no es grave, ya que pueden reescribirse como **functions** de tipo *Boolean (Logico)*; varias posibilidades para implementar mecanismos similares a las **secuencias** se explican mas abajo.

Se notara la ausencia de parametros por constante: Pascal/VS los permite, pero Pascal standard no.

Una limitacion importante del mecanismo de funciones en Pascal Standard (Pascal/VS no tiene esa limitacion) es que el tipo del resultado de una funcion ha de ser *entero, caracter, logico, real*, subrango, enumerado o pointer, pero no puede ser estructurado. Ello obliga a escribir lo que deberian ser funciones como procedures con parametros por variable: un procedure que sume matrices habra de formularse como

```
procedure Suma(a,b: matriz; var c: matriz);
```

y dejara en *c* el resultado de sumar *a* y *b*.

Las *directivas* dependen del compilador, y pueden servir para indicar que un subprograma esta escrito en otro lenguaje (p. ej., FORTRAN) u otras condiciones. Pascal Standard define una unica directiva (aunque preve mas): "FORWARD" despues de una cabecera de subprograma es el modo de "anunciar" o definir un subprograma, cuya implementacion se escribira mas abajo (obligatorio con subprogramas mutuamente recursivos); al escribir la implementacion, *deben omitirse tanto la lista de parametros (si la hay) como el tipo del resultado*:

```
procedure A(x: entero); forward;

procedure B(x: entero);
  A(p)
end (* B *);

procedure A: (* En este caso, SIN parametros *)
  B(q)
end (* A *);
```

Pascal/VS define otras directivas: EXTERNAL y ENTRY se estudiaran mas abajo; FORTRAN declara un subprograma como escrito en FORTRAN, y permite utilizar las rutinas escritas en ese lenguaje desde Pascal/VS.

Expresiones

```
expresion = expresion_simple [ operador_relacional expresion_simple]

expresion_simple = [signo] termino {operador_aditivo termino}

termino = factor {operador_multiplicativo termino}

factor = variable
  | constante_sin_signo
  | invocacion_a_funcion
  | conjunto
  | "(" expresion ")"
  | not factor

constante_sin_signo = numero_sin_signo
  | tira_de_caracteres
  | identificador_de_constante
  | nil

conjunto = "[" designacion_de_miembro {, designacion_de_miembro} "]"

designacion_de_miembro = expresion [.. expresion]

operador_relacional = in | < | > | = | ≤ | ≥ | ≠

operador_aditivo = + | - | or

operador_multiplicativo = * | / | div | mod | and

signo = + | -
```

Figura 128. Sintaxis de las expresiones en Pascal

Las expresiones en Pascal funcionan de modo distinto que en UBL:

- por un lado, no existen las construcciones
  - **y entonces**
  - **o sino**
  - **existe** ...
  - conversiones de tipo (en Pascal/VS si las hay)
- por otro, las prioridades de los operadores son distintas (o, si se prefiere, es distinta la sintaxis de las expresiones):
  - Los conjuntos se encierran entre corchetes en vez de entre llaves; puede utilizarse la notacion

    $expr_1$ .. $expr_2$

    para indicar el rango (que puede ser vacio) de los valores entre $expr_1$ y $expr_2$.
  - La prioridad de los operadores logicos obliga a encerrar las expresiones sometidas a un **o** o **y** logicos entre parentesis:

    (a < b) **o** *(c < d)*

    Puede utilizarse la siguiente regla: cada vez que se use un y o un **o** logico, a menos que los operandos sean factores (esto es, variables, constantes o invocaciones a funcion), deberan

encerrarse entre parentesis. Se notara, ademas, que pueden utilizarse operadores logicos distintos al mismo nivel, como en

b1 **or** *b2* **and** *b3*

(cosa que no puede hacerse en UBL), dado que **or** tiene la prioridad de "+" y **and** la de "*". La anterior expresion equivale a

b1 **or** *(b2* **and** *b3)*

El mecanismo de "packing" y las tiras de caracteres

Pascal permite manipular la representacion interna de los datos en el ordenador, especificando el prefijo **packed** (*empaquetado*) en un tipo **array**, **record**, **file** o **set**:

type *racional* = **packed record** *num,den: 1..10000* **end**;

"**packed**" es una indicacion al interprete para que utilice menos memoria para almacenar los objetos del tipo que es **packed**. Esto puede conseguirse, en muchos casos, al precio de disminuir la valocidad de ejecucion, y con ello la eficiencia del programa, pero es muy util al tratar estructuras de datos (como matrices) muy grandes. Para comprender el mecanismo de empaquetado, puede considerarse la representacion binaria de un entero (la mayoria de ordenadores representan los enteros en binario): un entero puede utilizar un numero fijo (digamos 16, o 32) de digitos binarios en su representacion, pero si sabemos (por ejemplo, gracias a una declaracion de subrango) que variara solo entre -100 y 100, basta con utilizar 8.

100 = 0000 0000 0000 0000 0000 0000 0110 0100 (con 32 digitos)

100 = 0110 0100 (con 8 digitos)

El caso de un objeto declarado como

packed array *[1..n]* **of** *char;*

donde *n* es una constante entera, tiene un significado especial en Pascal: dado que ese lenguaje no define un tipo *tira*, se considera que los **packed arrays of** *char* son tiras de caracteres *de longitud fija*, que las tiras literales (encerradas entre apostrofes) como

```
'Pagina 1'
```

son constantes de uno de tales tipos (con *n* = longitud de la tira), y que tiras de longitud distinta son siempre de tipos distintos.

Pascal/VS define un tipo *string(expresion_constante)* similar al *tira* de UBL. La unica direrencia es que los literales se encierran entre apostrofes.

Sintaxis general de un programa

```
program identificador ["("parametros_del_programa")"];
  declaraciones
begin
  {instruccion}
end.
```

Figura 129. ***Sintaxis de un programa en Pascal:*** Notese el punto despues de **end** que termina el programa.

Los parametros del programa son identificadores de objetos definidos externamente al programa; su significado depende del compilador: en el caso de Pascal/VS, se ignora. El identificador que da su nombre al programa puede ser arbitrario, y no se utiliza para nada dentro de este; en Pascal/VS, debe coincidir con el nombre del fichero que contiene el programa.

Entrada y salida en Pascal/VS

Los mecanismos de entrada y salida en Pascal VS extienden los de Pascal en dos direcciones: ampliacion del uso de formatos y "conversiones internas" mediante lectura/escritura sobre tiras de caracteres.

Pascal VS permite utilizar formatos en lectura (cosa que Pascal ISO no hace), y da una interpretacion a los formatos negativos: se utilizan como indicacion de que hay que alinear a la izquierda, en vez de a la derecha.

Comparense, como ejemplo, los resultados producidos por las dos instrucciones siguientes:

```
for i := 1 to 5 do writeln('''',i*i*i:3,'''');
```

que escribe

```
'  1'
'  8'
' 27'
' 64'
'125'
```

y

```
for i := 1 to 5 do writeln('''',i*i*i:-3,'''');
```

que escribe

```
'1  '
'8  '
'27 '
'64 '
'125'
```

Los procedimientos predefinidos *readstr* y *writestr* realizan entrada y salida *sobre tiras de caracteres* en vez de sobre filas. El primer argumento debe ser una tira de caracteres, sobre o desde la cual se manejaran los datos.

Por ejemplo,

```
writestr (s,4:3,'*':-2,s)
```

añade por la izquierda la tira

```
"  4* "
```

al contenido anterior de s.

Compilacion externa en Pascal/VS

Distinguiremos la *compilacion separada* de la *compilacion externa* del siguiente modo: llamaremos compilacion separada a algun mecanismo que permita compilar un programa "a trozos" logicamente dependientes, o un subprograma o un modulo por si mismo, de modo que, aparte del hecho de que los trozos residan en ficheros distintos, el resultado de la compilacion, en cuanto a verificacion de tipos y espacios de nombres disponibles, sea el mismo que si no se utilizase la compilacion separada; y compilacion externa, a cualquier mecanismo que implemente algun subconjunto de la compilacion separada, a base de restringir la generalidad de este.

Pascal/VS proporciona un mecanismo de compilacion externa, junto con algunas construcciones mas o menos anecdoticas y dependientes de sistema para solventar los problemas que la reduccion con respecto a la idea de compilacion separada presenta.

Una *unidad de compilacion* o fichero compilable puede ser un **programa** o bien un *SEGMENTo*. Un segmento tiene la siguiente sintaxis

```
segmento =
  SEGMENT identificador ;
  declaracion
  {declaracion}
  .
```

Figura 130. Sintaxis de un 'segment' en Pascal/VS: SEGMENT no es una palabra reservada; notese tambien el punto final que termina el segmento.

y puede contener cualquier tipo de declaracion, excepto declaraciones de etiquetas (**label**). Los subprogramas que se declaran (el nombre de alguno de los cuales puede ser identico al nombre del segmento) pueden utilizarse desde el programa principal o desde otro segmento si alli estan declarados como "EXTERNAL", como en

```
function raiz(x: real): real; external;
```

y, en el segmento, como "ENTRY":

```
function raiz(x: real): real; entry;
```

Ademas, pueden declararse variables; si estas variables coinciden en su orden y sus tipos con las declaradas en el programa, se supone que los subprogramas del segmento pueden acceder a ellas (aunque el compilador no verifica estas equivalencias, que pueden por tanto no cumplirse, produciendo asi errores dificiles de detectar). Un metodo mas seguro para el intercambio de variables lo proporcionan las variables **def** y **ref**: si una variable se declara con la palabra reservada **ref** (o **def**) en vez de **var**, puede ser definida y utilizada en cualquier otro subprograma (de la misma unidad de compilacion o no) contando con que tales definiciones se refieren al mismo objeto. Estos objetos son *estaticos*, en el sentido de que se crean al compilar el programa y no al crear cada instancia del subprograma que las define -- si es que hay alguno, pues tambien pueden estar declaradas directamente en un segmento. Ello implica, entre otras cosas, que no se crean copias nuevas si se realiza una invocacion recursiva, y que puede confiarse en que retienen su valor entre invocaciones de un subprograma -- o entre "visitas" a un segmento. Estas anomalias aconsejan confinar su uso a aquel para el que se supone que fueron diseñadas: compartir objetos entre distintas unidades de compilacion. Una declaracion **def** *define* un objeto compartible; una declaracion **ref** lo *referencia*. Por tanto, el objeto debe ser definido en un unico lugar como **def**, y en los demas como **ref**.

Una variable tambien puede declararse como **static**, con lo que pasa a funcionar como una **def**, pero sin poder referirse a ella mas que del modo usual.

En el caso de variables **static** o **def**, existe un metodo util para darles valores iniciales: la declaracion de **value**:

```
static i: integer;
value i := 3;
```

Se escribe "**value**" seguido de una asignacion, que debe contener solo referencias estaticamente evaluables [esto es, la parte izquierda debe ser una objeto o un campo de una tupla o un elemento (de subindice constante) de una tabla definida como **static** o **def**; y la parte derecha debe ser una expresion constante].

Expresiones constantes en Pascal/VS

Ademas de permitir combinaciones de constantes donde Pascal ISO requiere constantes, Pascal/VS permite el uso (no necesariamente en declaraciones **value**) de *constantes estructuradas*, que se forman precediendo la lista entre parentesis de los valores de las componentes del objeto estructurado (**set**, **array** o **record**) con el nombre de su tipo:

```
type complejo = record re,im: real end;

const i = complejo(0,1);

var c: complejo;
```

Pueden omitirse valores en la lista (en cuyo caso la componente correspondiente queda indefinida

```
c := complejo(,3); (* c.re indefinido *)
```

Como tambien puede no escribirse el nombre del tipo cuando se encuentra una constante estructurada dentro de otra, como en

```
type t = array [1..2] of complejo;

const t1 = t(complejo(0,1),complejo(1,0));
const t2 = t((0,1),(1,0));
```

donde *t1* y *t2* son dos modos de escribir la misma constante.

Instrucciones '%' del compilador de Pascal/VS

El compilador de Pascal/VS reconoce una especie de *pseudo-comentarios*, que empiezan con el caracter "%" y terminan con el fin de linea, y cuya funcion es informar al compilador de determinadas condiciones, tales como si se desea o no que verifique los posibles errores de ejecucion, o el aspecto que ha de tomar el listado. Se permiten las siguientes posibilidades:

% INCLUDE **identificador** ["("**identificador**")"] causa la inclusion del fichero identificado por el nombre de libreria (antes del parentesis: o "SYSLIB" si no hay parentesis) y el nombre de miembro (entre parentesis: o el identificador si no hay parentesis) como si formase parte del texto de la unidad de compilacion en el mismo lugar donde se encuentra la instruccion %INCLUDE.

% CHECK **opcion** (ON | OFF) donde

opcion = POINTER | SUBSCRIPT | SUBRANGE | FUNCTION | CASE | TRUNCATE

permite activar selectivamente las comprobaciones que el compilador hace para ver si se produce algun error en el uso del pointer **nil** como nombre de algun objeto (POINTER), al subindiciar una tabla con un subindice incorrecto (SUBSCRIPT), al asignar a un objeto de tipo subrango un valor extraño al rango (SUBRANGE), al terminar de ejecutar una **function** sin darle valor (FUNCTION), al ejecutar una instruccion **case** sin que el valor de la expresion coincida con alguna etiqueta (CASE) o, en el manejo de tiras, al intentar asignar una tira demasiado larga a un objeto de tipo tira (TRUNCATE).

% PRINT (ON | OFF) PRINT OFF suprime el listado del programa hasta el final o hasta que se encuentre PRINT ON.

% LIST (ON | OFF) Activa o suprime el listado de la traduccion del programa a lenguaje ASSEMBLER.

% PAGE provoca que la siguiente linea sea impresa al principio de la siguiente pagina

% CPAGE **entero** Pasa de pagina solo si quedan en la actual menos de "entero" lineas.

% TITLE **tira de caracteres** Pasa pagina y cambia el titulo que aparece a principio de pagina por "tira de caracteres".

% SKIP **entero** Deja "entero" lineas en blanco en el listado

% MARGINS **entero1 entero2** Provoca que el compilador solo interprete el texto comprendido entre las columnas "entero1" y "entero2" del fichero (por defecto, se toman las columnas 1 y 72).

Traduccion de las posibilidades de UBL que no ofrece Pascal

Tiras de caracteres

El mecanismo de tiras en Pascal Standard es bastante limitado (Pascal/VS permite un tipo *string(longitud)* similar al *tira* de UBL): se considera que cada literal de tira, de la forma

'dos o mas caracteres entre apostrofes'

tiene tipo

packed array *[1..l]* **of** *char;*

donde *l* es la longitud de la tira; se permite tambien la escritura de objetos (no necesariamente constantes) de esos tipos. Observese que, asi definidas, las tiras son siempre de *longitud constante*: declarado un objeto como tira de 8 caracteres, por ejemplo, solo es posible asignarle valores de su mismo tipo: esto es, si son constantes, literales de exactamente ocho caracteres (con blancos a la derecha si es necesario).

Aunque casi todos los compiladores ofrecen alguna extension que permite el manejo de tiras de longitud variable, tales variaciones no se ajustan al Standard, y por tanto no suelen ser portables. Si se desea obtener el efecto de tiras de longitud variable mediante los **packed arrays** de caracteres, debe cuidarse de rellenar las posiciones "sobrantes" de la tabla con espacios en blanco, y controlar la longitud de la tira mediante un objeto separado que sea un subindice sobre la tira.

Secuencias

Dado que el lenguaje Pascal carece del concepto de **secuencia**, la mejor forma de programar en Pascal es diseñando los programas sin recurrir a ese concepto; de todos modos, si fuese necesario adaptar un programa ya escrito en UBL, las dos tecnicas que siguen podrian ser de utilidad.

Una secuencia es una accion con una accion como parametro: Consideremos, por ejemplo, la **secuencia**

```
secuencia S(i: entero): entero es
  var j: entero;
haz
  j ← 1;
  mientras j < i haz
    produce j*j;
    j ← j + 1;
  fin mientras;
fin S;
```

que produce los cuadrados de los enteros menores que su argumento, y una utilizacion cualquiera de ella:

```
para k en s(p) haz
  escribe_linea k;
fin para;
```

Para eliminar la secuencia (y asi aumentar la traducibilidad del programa) podemos reescribir las instrucciones subordinadas a la instruccion **para** en forma de **accion**, parametrizada por la variable de control:

```
accion Acc(k: entero) haz
  escribe_linea k;
fin Acc;
```

la secuencia, tambien como una **accion**, con un parametro mas, "compatible" con la anterior **accion**,

```
accion S(i: entero;accion A(k: entero)) es
  var j: entero;
haz
  j ← 1;
  mientras j < i haz
    A(j*j);
    j ← j + 1;
  fin mientras;
fin S;
```

donde hemos substituido cada instruccion **produce** por una invocacion al nuevo subprograma parametrico; por ultimo, la instruccion **para** completa se reemplazara por

```
S(p,Acc);
```

con lo que se obtendra el efecto deseado, sin emplear secuencias. Este metodo, aunque tedioso, es absolutamente general, y puede aplicarse a toda clase de secuencias, aun recursivas; el que sigue es mas limitado, pero mas efectivo en los casos en los que puede utilizarse:

Una secuencia es un artefacto con varios controles u operaciones:: Se trata de reescribir la secuencia, al modo de un **modulo**, en forma de 5 subprogramas mediante los que se realiza una inspeccion similar a la de las **filas**

- **accion** *activa*, que "pone en marcha" la secuencia;
- **condicion** *mas*, que indica si quedan valores en la secuencia
- **funcion** *valor: tipo*, si *mas*, es el "valor actual" de la secuencia
- **accion** *siguiente*, que "pasa" al siguiente valor de la secuencia; y
- **accion** *desactiva*, que "termina" la secuencia.

para luego traducir una instruccion **para**

```
para c en S haz I; fin;
```

por

```
activa;
mientras mas haz
  c ← valor;
  I;
  siguiente;
fin mientras;
desactiva;
```

Damos una ejemplo de esta descomposicion, aplicado a la anterior secuencia:

```
modulo Secuencia_S es
  accion Activa(i: entero); -- toma como parametro el de la secuencia
  accion Desactiva;
  condicion Mas;
  funcion Valor: entero;
  accion Siguiente;
fin Secuencia_s;

modulo implementa Secuencia_S es
  var II,j: entero
  accion Activa(i: entero) haz II ← i; fin;
  accion Desactiva haz nada; fin;
  condicion Mas haz vale j < II; fin;
  funcion Valor: entero haz vale j * j; fin;
  accion Siguiente haz j ← j + 1; fin;
haz
  j ← 1;
fin Secuencia_s;

usa Secuencia_s;
```

La instruccion **para** se escribe, dado ese modulo, como

```
activa;
mientras mas haz
  k ← valor;
  escribe_linea k;
  siguiente;
fin mientras;
desactiva;
```

Tabla de palabras reservadas

and			* static
array	file	of	
* assert	for	or	then
	function	* otherwise	to
begin			type
	goto	packed	
case		procedure	
const	if	program	until
* continue	in		
		* range	* value
* def	label	record	var
div	* leave	* ref	
do		repeat	while
downto	mod	* return	with
else	nil	set	
end	not	* space	* xor

Figura 131. Tabla de identificadores reservados en Pascal: las palabras señaladas por un asterisco solo son reservadas en Pascal/VS.

Ejemplo

Como ejemplo presentamos la siguiente reescritura del programa *analiza_frecuencias* en el Algoritmo 48 en la pagina 147 y sus correspondientes implementaciones. Se notara el modo en que se ha escrito lo que en UBL era una **secuencia**.

```
program ejemplo;
 label Fin_normal, Fin_erroneo;
 type clave = string(10);
 var ch: char; (* ultimo caracter leido *) c: clave;
 function valor(const c: clave): integer; external;
 function esta(const c: clave): boolean; external;
 function cabenmas: boolean; external;
 procedure davalor(const c: clave; i: integer); external;
 procedure pasea (procedure p(const c: clave; const i: integer)); external;

 procedure lee_tira(var c: clave); begin
  c := ''; while not eof and (ch = ' ') do read(ch);
  if not eof then while ch <> ' ' dobegin c := c || str(ch); read(ch) end
 end (* lee_tira *);

 procedure estadistica (const c: clave; const i: integer); begin
  writeln('La palabra ''',c,''' aparece ',i:1,' veces.')
 end;

begin
 read(ch);
 while true do begin (* forever *)
  lee_tira(c);
 if c = '' then goto Fin_normal;
  if esta(c) then (* incrementa frecuencia *) davalor(c,valor(c) + 1)
  else if cabenmas then (* crea frecuencia *) davalor(c,1)
  else begin (* no caben mas *) writeln('Error en la tabla de frecuencias'); goto Fin_erroneo
  end (* if *)
 end (* do forever *);
Fin_normal:
 pasea(estadistica);
Fin_erroneo:
end.
```

Algoritmo 55. Programa de ejemplo en Pascal/VS: frecuencias de palabras en un texto

```
segment segm1;

 type clave = string(10);

 def t: array[1..100] of packed record ant,sig: 0..100; cl: clave; n: integer end;

 def libre: 1..101; value libre := 1;

 function valor(const c: clave): integer; entry;
  var act: 0..100;
 begin
  if libre > 1 then begin
   act := 1;
   repeat
    with t[act] do
     if cl = c then begin valor := n; return end
     else if cl < c then act := sig else act := ant
   until act = 0
  end
  (* Error : clave erronea *)
 end (* valor *);

 function esta(const c: clave): boolean; entry;
  var act: 0..100;
 begin
  if libre > 1 then begin
   act := 1;
   repeat
    with t[act] do
```

```
      if cl = c then begin esta := true; return end
      else if cl < c then act := sig else act := ant
    until act = 0
  end;
  esta := false
end (* esta *);

function cabenmas: boolean; entry; begin cabenmas := libre ≤ 100 end;

procedure davalor(const c: clave; i: integer); entry;
  var act,antr: 0..100; izq: boolean;
begin
  if libre = 1 then begin
    libre := 2;
    with t[1] do begin cl := c; sig := 0; ant := 0; n := i end;
    return
  end else begin
    act := 1; antr := 1;
    repeat
      with t[act] do
        if cl = c then begin n := i; return end
        else begin
          antr := act;
          if cl < c then begin act := sig; izq := false
          end else begin act := ant; izq := true
          end
        end (* if *)
    until act = 0
  end;
  libre := libre + 1;
  if izq then t[antr].ant := libre else t[antr].sig := libre;
  with t[libre] do begin cl := c; n := i; ant := 0; sig := 0 end end (* esta *);

procedure pasea (procedure p(const c: clave; const i: integer)); entry;
  procedure pasea2(i: integer; procedure p(const c: clave; const i: integer));
  begin
    if i ≠ 0 then with t[i] do begin
      pasea2(ant,p);
      p(cl,n);
      pasea2(sig,p)
    end (* if *)
  end (* pasea2 *);
begin
  if libre > 1 then pasea2(1,p)
end (* pasea *);
```

Algoritmo 56. Implementacion de una tabla de frecuencias utilizando vectores

```
segment segm2;

  type clave = string(10);

  type arbol = ↑ tupla_arbol;
  type tupla_arbol = record ant,sig: arbol; cl: clave; n: integer end;

  def raiz: arbol; value raiz := nil;

  function valor(const c: clave): integer; entry;
    var act: arbol;
  begin
    if raiz ≠ nil then begin
      act := raiz;
      repeat
```

```
      with act↑ do
        if cl = c then begin valor := n; return end
        else if cl < c then act := sig else act := ant
      until act = nil
    end
    (* Error : clave erronea *)
  end (* valor *);

  function esta(const c: clave): boolean; entry;
    var act: arbol;
  begin
    if raiz ≠ nil then begin
      act := raiz;
      repeat
        with act↑ do
          if cl = c then begin esta := true; return end
          else if cl < c then act := sig else act := ant
      until act = nil
    end;
    esta := false
  end (* esta *);

  function cabenmas: boolean; entry; begin cabenmas := true end;

  procedure davalor(const c: clave; i: integer); entry;
    var act,antr: arbol; izq: boolean;
  begin
    if raiz = nil then begin
      new(raiz);
      with raiz↑ do begin cl := c; sig := nil; ant := nil; n := i end;
      return
    end else begin
      act := raiz; antr := raiz;
      repeat
        with act↑ do
          if cl = c then begin n := i; return end
          else begin
            antr := act;
            if cl < c then begin act := sig; izq := false
            end else begin act := ant; izq := true
            end
          end (* if *)
      until act = nil
    end;
    new(act);
    if izq then antr↑.ant := act else antr↑.sig := act;
    with act↑ do begin cl := c; n := i; ant := nil; sig := nil end end (* esta *);

  procedure pasea (procedure p(const c: clave; const i: integer)); entry;
    procedure pasea2(a: arbol; procedure p(const c: clave; const i: integer));
    begin
      if a ≠ nil then with a↑ do begin
        pasea2(ant,p);
        p(cl,n);
        pasea2(sig,p)
      end (* if *)
    end (* pasea2 *);
  begin
    pasea2(raiz,p)
  end (* pasea *);
```

Algoritmo 57. Implementacion de una tabla de frecuencias utilizando pointers

Apendice 2: El compilador UBL/CMS version 0.1

Estructura general

El compilador UBL/CMS es un prototipo experimental desarrollado a partir del sistema PASCAL P4 (Nori, Amman y otros, E.T.H., Zurich, 1976: ese sistema ha servido de base a compiladores como UCSD Pascal o IBM Pascal/VS), mediante extensiones incorporadas por el metodo de "bootstrapping". El compilador consta de dos fases: en la primera, se realiza el analisis sintactico y se genera un listado de compilacion y un fichero que contiene codigo (PCODE) para una version modificada del "Stack Computer" descrito en la informacion distribuida con Pascal P4; en la segunda, se ensambla el codigo intermedio (PCODE) en formato compatible con los cargadores de los sistemas operativos para IBM/370 y similares. Ese codigo se ejecuta con la ayuda de un interprete y de una libreria de programas para la ejecucion.

El caracter de prototipo del compilador, asi como la interpretacion del codigo generado, hacen que la eficiencia sea considerablemente menor que la de los lenguajes y compiladores normalmente disponibles; esta es una caracteristica de la implementacion descrita, y en ningun caso debe entenderse como señal de algun tipo de ineficiencia intrinseca del lenguaje UBL.

El siguiente diagrama muestra la estructura del compilador UBL/CMS y de los ficheros generados.

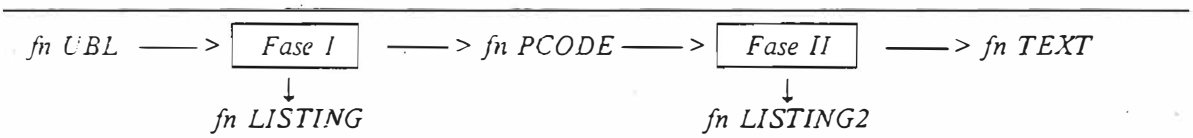


Figura 132. ***Estructura del compilador UBL/CMS:*** los ficheros de tipo PCODE y LISTING2 son destruidos despues de la compilacion (a menos que se especifique lo contrario): asi, el compilador produce un LISTING y un TEXT, como sucede el la mayoria de los demas lenguajes.

Representacion interna de los datos

La representacion interna de los datos se ha escogido para facilitar la compilacion, y en algunos casos (p. ej., en los valores enumerados y en las tiras de caracteres) supone un claro desperdicio de memoria.

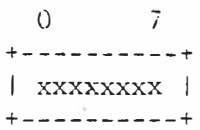
Representacion de los valores de tipo *logico*

Se representan utilizando un byte: el bit situado mas a la derecha es 1 si el valor es *cierto*, y 0 si es *falso*; los demas bits deben ser siempre 0.

```
  0       7
+----------+
| 0000000x |   x = 1 → CIERTO, x = 0 → FALSO
+----------+
```

Representacion de los valores de tipo *caracter*

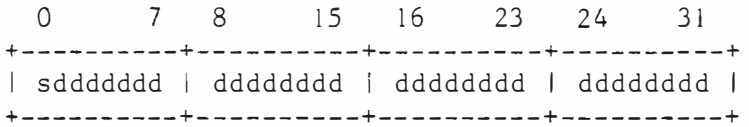
Se representan en el codigo EBCDIC utilizando un byte.



Las letras no son contiguas en EBCDIC (esto es, que *C* sea de tipo *['a'..'z'] no* implica que *C* sea una letra).

Representacion de los valores de tipo *entero* o enumerado

Los enumerados se representan almacenando su ordinal; los enteros, como una palabra con signo en complemento a dos de 32 bits (4 bytes).




```
S es el signo: s = 0 → positivo, s = 1 → negativo
```

Los digitos D representan el numero en binario, si es positivo, o el complemento logico de su valor absoluto, mas uno, si es negativo: por ejemplo, el entero -5 se representa del siguiente modo:

```
S = 1, ya que -5 es negativo; para calcular los digitos D,

5 = 101 (en binario) =
  0000000 00000000 00000000 00000101

Su complemento es
  1111111 11111111 11111111 11111010

y sumando 1 se obtiene
  1111111 11111111 11111111 11111011

Si por ultimo añadimos el bit S de signo a la izquierda, se
obtiene la representacion buscada:

  11111111 11111111 11111111 11111011
```

Los limites a los valores enteros impuestos por esta representacion son

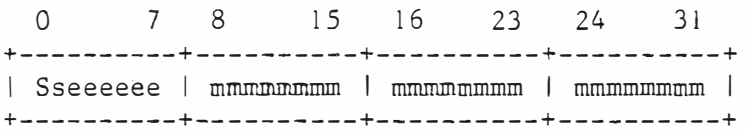
Para todo n *entero, -2147483648* ≤ n ≤ *2147483647*

No se usan enteros de 3, 2 o 1 bytes.

Los enteros y enumerados se alinean a frontera de 4 bytes.

Representacion de los valores de tipo *real*

Se utiliza el formato corto de punto flotante de los sistemas IBM/370, con mantisa de 24 bits, exponente en base 16 de 6 bits, dos bits de signo y normalizacion a base 16:




```
S: signo del numero
s: signo del exponente
eeeeee: exponente (potencia de 16)
mmmm... : mantisa (el primer m, distinto de 0)
```

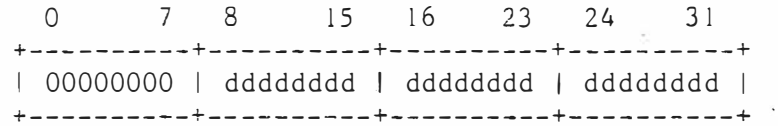
Con esta representacion. se obtiene una precision algo superior a los seis digitos decimales. y un rango de exponentes situado entre -74 y 76. aproximadamente.

Los reales se alinean a frontera de 4 bytes.

No se implementan reales de precision doble (REAL*8 o DOUBLE PRECISION de FORTRAN) ni cuadruple (REAL*16).

Representacion de los valores de tipo nombre

Se utilizan direcciones de 24 bits, extendidas a la izquierda con ceros hasta llegar a 32 bits. El valor **nulo** se representa como 32 ceros.

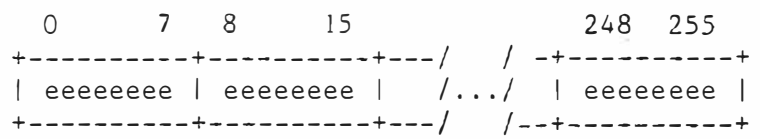


ddd... es la direccion del objeto nombrado.

Los nombres se alinean a frontera de 4 bytes.

Representacion de los valores de tipo conjunto

Se utilizan 32 bytes (256 bits) para representar la funcion caracteristica del conjunto. Si se trata de un conjunto de enumerados o caracteres, se toman sus ordinales: asi puede considerarse que todos los conjuntos son de enteros.



e = 1 si el elemento esta en el conjunto
e = 0 si el elemento no esta en el conjunto

Los ordinales maximo y minimo del tipo base de un conjunto deben estar comprendidos entre 0 y 255.

Se consideran los bits numerados de 0 a 255; cada bit es 1 si el elemento correspondiente esta en el conjunto, y 0 si no esta (en el caso de conjuntos de menos de 256 elementos, los bits sobrantes son siempre 0).

Los conjuntos se alinean a frontera de 1 byte.

Representacion de objetos estructurados

Las **tablas** y las **tuplas** se forman añadiendo los elementos constituyentes por la derecha, con relleno si es preciso para conseguir la alineacion correcta.

Las tiras de caracteres se representan como una tabla de caracteres cuya longitud es la maxima declarada precedida de un campo entero de 32 bits que indica su longitud actual. Su alineamiento es de 4 bytes.

El compilador no implementa **filas**, excepto las predefinidas *entrada* y *salida* y dos mas, tambien predefinidas y auxiliares, llamadas *entrada2* (en modo de entrada) y *salida2* (en modo de salida). Las caracteristicas de esas filas son fijas:

fila	*ddname*	*recfm*	*lrecl*
entrada	*INPUT*	*F*	*80*
salida	*OUTPUT*	*F*	*121*
entrada2	*PRD*	*F*	*80*
salida2	*PRR*	*F*	*80*

por defecto, estan asignadas a la terminal, aunque pueden reasignarse mediante la instruccion de CMS FILEDEF.

Las acciones predefinidas *conecta, desconecta, libera* no estan implementadas.

Instrucciones '%'

El compilador admite instrucciones que comienzan con el simbolo "%" para controlar la apariencia del listado. cuales columnas son significativas para la compilacion, etc.

Pueden utilizarse las siguientes:

% PAGINA provoca un salto de pagina en el listado: la siguiente linea sera la primera de la siguiente pagina.

% TITULO tira de caracteres los caracteres siguientes al blanco que sigue a la palabra "titulo" pasan a imprimirse en la primera linea de cada pagina del listado. La instruccion fuerza tambien un cambio de pagina.

% MARGENES m n indica que las columnas significativas son las comprendidas entre las columnas *m* y *n* (inclusive): las restantes columnas se toman como comentarios.

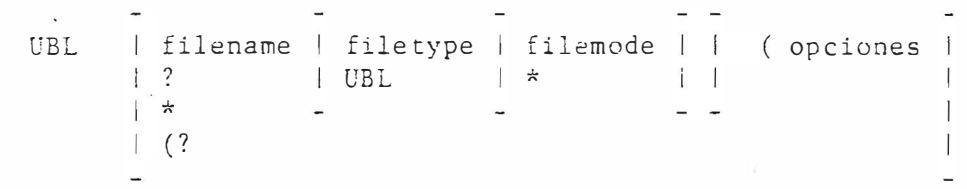
% INCLUYE filename opera como si, en el lugar en que se encuentra la instruccion "incluye", se hallase el texto correspondiente al fichero de nombre *filename* y tipo UBL.

Si la instruccion es correcta, no aparece en el listado.

Utilizacion: instrucciones de CMS

Compilacion

Los ficheros que contienen programas en UBL deben ser de longitud fija y de 80 caracteres por linea. Se compilan mediante la instruccion (EXEC) de CMS UBL, cuyo formato se describe a continuacion:




```
opciones: CAStellano | CATala
          Source | NOSource
          HOld | NOHold | PCode
          UNder | NOUNder | SUBR
          OVer | NOOVer | NEGR
          DIsk | PRint | NOPRint
          OBJect | NOOBJect
          Debug | NODebug | ND
```

Figura 133. Formato de la instruccion UBL: la parte de las opciones que aparece en minusculas es opcional.

"UBL ?" proporciona informacion interactiva sobre el formato y funcionamiento de la instruccion UBL; "UBL (?" lo hace con las opciones de compilacion.

A continuacion se describe cada una de las opciones:

- Si se usa '*' en vez de un identificador de fichero, pueden especificarse 'opciones por defecto' de las compilaciones subsiguientes. Este metodo es acumulativo: las opciones de una instruccion 'ubl * (opciones' se añaden a las de anteriores instrucciones similares. Estas opciones se pierden al hacer LOGOFF o IPL.

- CASTELLA ¦ CASTELLANO ¦ CATALA identifica el idioma en el que esta escrito el programa que se desea compilar. Debe especificarse esta opcion; de lo contrario, se supone que el programa esta en ingles.

- SOURCE ¦ NOSOURCE decide si el listado producido por el compilador contendra una reproduccion del programa fuente. Por defecto se supone SOURCE.

- HOLD ¦ HO ¦ NOHOLD ¦ NOH ¦ PCODE decide si se conservan o no los ficheros de trabajo (de tipo PCODE y LISTING2) utilizados por el compilador. Por defecto se supone NOHOLD.

- UNDER ¦ NOUNDER ¦ SUBR especifica si se desea o no que las palabras reservadas aparezcan subrayadas en el listado. Se supone NOUNDER por defecto.

- OVER ¦ NOOVER ¦ NEGR especifica si se desea o no que las palabras reservadas aparezcan en negrita en el listado. Se supone NOOVER por defecto.

- DISK ¦ PRINT ¦ NOPRINT especifica el destino del listado: fichero en disco, impresora virtual del usuario o no hay listado. Se supone DISK por defecto.

- OBJECT ' NOOBJECT especifica si se desea que el compilador genere o no un fichero TEXT conteniendo la traduccion del programa. Por defecto se supone OBJECT.

- DEBUG ¦ NODEBUG especifica si se desea que el programa contenga instrucciones que verifiquen la ocurrencia de errores en tiempo de ejecucion o no. Se supone DEBUG por defecto.

En el listado aparece el programa, posiblemente intensificado (de acuerdo con las opciones NEGR o SUBR), limitado a la izquierda por una linea vertical y por arriba mediante una linea de escala. A la izquierda de la linea vertical aparecen numeraciones de las lineas del programa (bajo el titulo "Linea") y de las instrucciones (bajo el titulo "Sentencia"). Las instrucciones "%" no aparecen en el listado, si son correctas. Cada linea con errores esta seguida de indicaciones de las posiciones en las que ocurren los errores y de un numero que identifica el error, asi como de lineas adicionales explicando la causa de los errores. Debido a la naturaleza monopaso del compilador, es posible que algunos errores se diagnostiquen como ocurriendo en el simbolo lexico siguiente a aquel en el que realmente ocurren; tambien es posible la aparicion de errores en cascada.

Ejecucion

Para ejecutar un programa compilado en UBL, se utilizara la instruccion RUBL:

```
RUBL filename
```

donde "filename" es el nombre del fichero que se compilo. No debe utilizarse la secuencia habitual de instrucciones LOAD y START, ni RUN. Tampoco es necesario hacer ningun GLOBAL, aunque quiza si algun FILEDEF (antes de RUBL) si los ficheros que se usan no son la terminal.

Indice Tematico

F

G

I

L

M

N

O

P

R

S

T

U

V