INTRODUCCION A LA PROGRAMACION EN UBL

INTRODUCCION A LA PROGRAMACION EN UBL

INTRODUCCION A LA PROGRAMACION EN UBL

J.M. BLASCO

CENTRE D'INFORMÀTICA · UNIVERSITAT DE BARCELONA

PPU · Promociones Publicaciones Universitarias

Edita: Centro de Informática. Universidad de Barcelona
PPU. Promociones Publicaciones Universitarias

Imprime: Novo-grafic. Riera San Miquel, 3. Barcelona

Dep. Legal: B-13018-85
I.S.B.N. 84-86130-92-1

Prefacio

"Programming is one of the most difficult branches of applied Mathematics, the poorer mathematicians had better remain pure mathematicians."
(EDSGER W. DIJKSTRA, en *Selected Writings on Computing: A Personal Perspective*)

"Aprenda BASIC *en 7 días."*
(De los anuncios)

"(...) Hemos definido pues el concepto de mapping, *introduciendo la distinción entre* partial *y* total mapping; *en el caso de que el* origin set *corresponda con el* domain set, *(...)"*
(Fragmento de una clase de Matemáticas imaginaria)

Presentación

Cada día más personas desean utilizar ordenadores en el ámbito de la investigación científica universitaria. Dicho muy rápido, se puede distinguir entre dos tipos de tal utilización: por un lado, el uso de "paquetes" de programas ya escritos, que proporcionan respuestas a rangos, a veces amplios pero siempre determinados, de problemas; por otro, la que sucede cuando es necesario programar el ordenador para afrontar problemas cuya solución no se conoce o para los cuales no se dispone de las soluciones conocidas. Este libro esta concebido para proporcionar una introducción a las técnicas básicas necesarias para afrontar una necesidad de este tipo.

Se suele pensar que aprender a programar consiste en llegar a saber cómo funciona un lenguaje. Así, muchos cursos de Programación se limitan a una enumeración descriptiva de "lo que se puede hacer con este lenguaje", dejando de lado o pretendiendo que se asimile por ósmosis lo que en resumidas cuentas es más difícil – de aprender y de explicar –: cómo se afronta un problema mediante un ordenador, cómo debe modelizarse la realidad hasta un punto suficientemente formal como para que pueda ser "digerido" por el ordenador, de qué métodos se dispone para llegar a tal formalización ...

Además, ¿con qué lenguaje se introduce la programación?. Los usuarios predominantemente "científicos"[1] opinarán que FORTRAN es el más adecuado; los orientados a aplicaciones de gestión se inclinarán por PL/I, o incluso por COBOL; quien trabaje con microordenadores anunciará la superioridad de BASIC; habrá defensores de PASCAL, y quizás alguno lo será de APL, LISP, ADA, PROLOG... Cada uno de esos lenguajes es distinto, en su filosofía y en sus posibilidades. Se acepta comunmente que la elección del lenguaje con el que se toma contacto por primera vez con el ordenador condiciona fuertemente[2] la visión que se tomará de la programación y la capacidad posterior de resolución de problemas. En este contexto, el lenguaje PASCAL parece el más apropiado para enseñar a programar, ya que fue diseñado con ese propósito e incluye las estructuras básicas de un modo elegante, claro y pedagógico; pero tiene ya más de 15 años (la primera versión es de 1968) y se ha avanzado bastante desde entonces: por un lado, en la línea de encontrar derivados de PASCAL que corrijan sus carencias e imperfecciones (e.g., MODULA, CLU, ALPHARD, EUCLID, y, en cierto sentido, ADA); por otro, desarrollando lenguajes enteramente nuevos (como PROLOG o SMALLTALK). Aunque observamos con interes la evolución de las nuevas estilos, creemos más conveniente para el usuario universitario la formación en un lenguaje imperativo convencional, dado que facilita el aprendizaje posterior de los lenguajes actualmente en uso. Y PASCAL es, a nuestro juicio, el más capacitado para soportar esa tarea.

Tiene, sin embargo (como todos los demás lenguajes), un inconveniente grave: está diseñado en inglés, y por tanto obliga a aprender los conceptos elementales de programación utilizando palabras inglesas. Los que estén algo familiarizados con el mundo de la Informática encontrarán esto normal (de hecho, casi todo en Informática está en inglés), pero, bien considerado no lo es tanto: equivale a recibir una clase sobre cualquier materia, para la que existe un vocabulario castellano (o catalán) conocido, en la que se haya substituido la nomenclatura por su equivalente inglesa, quizá con la explicación de que gran parte de la literatura sobre el tema se halla escrita en ese idioma. Por otra parte, estudios realizados sobre el tema[3] muestran que el aprendizaje de la programación utilizando la lengua materna facilita la asimilación al aumentar la mnemotecnia de los programas y evitar distracciones con conceptos extraños.

---

1 Nos referimos a las especialidades propias de Físicos, Matemáticos, Químicos, Ingenieros, ... Esta distinción es cuestionable, pero se incurre en ella con ánimo de simplificar la discusión.

2 Algunos dicen que puede llegar a deformar la mente: véase por ejemplo [Dijk 82].

3 *Cfr.* los trabajos de Elisabet Tubau *et al.* [Tub 84a], [Tub 84b], [Tub 84c], y la referencia [PBot 84], en la que se describe el lenguaje MERLIN, que responde a una filosofía similar a la aqui presentada, con el que vienen desarrollandose cursos introductorios en la Facultad de Informática de la UPC desde hace varios años.

Así, consideramos que un lenguaje que

1. sea derivado de PASCAL,

2. mejore y extienda ese lenguaje en las áreas que se han reconocido como modificables, y

3. sea lo más parecido posible a la lengua habitualmente utilizada, sin renunciar por ello al rigor necesario en la expresión de algoritmos.

será el mas adecuado para los fines que nos hemos propuesto.

El Lenguaje de la Universidad de Barcelona (UBL), que se describe y utiliza en este libro para enseñar a programar, intenta cumplir estos requisitos. Se ha utilizado durante dos años como soporte de la asignatura "Introducción a la Programación" del "Curso de Programador de Aplicaciones Científicas" realizado por el Centro de Informática de la Universidad de Barcelona, y es objeto de reelaboración continua sobre la base de la experiencia docente y de las investigaciones relacionadas.[4] Incorpora también, respecto al PASCAL, novedades tanto sintácticas (e.g., *todas* las instrucciones terminan en punto y coma; *todas* las instrucciones son parentizadas) como semánticas (e.g., módulos, sucesiones o enumerados no ordenados) reconocidas en la literatura informática[5] como convenientes.

Los lenguajes de programación no son lenguajes

La denominación "Lenguajes de Programación", aunque comunmente aceptada, es más bien nefasta: induce a creer que las notaciones algoritmicas son realmente lenguajes;[6] por tanto, sólo habrá que "aprenderlas", y luego "comunicarse con el ordenador" para "decirle lo que queremos", todo lo cual es totalmente erróneo.

Aunque es difícil eliminar, en un primer nivel, las referencias intuitivas de este tipo, que pueden ser útiles como metáforas para conseguir una mejor comprensión de los mecanismos involucrados en el uso de notaciones algoritmicas, aqui hemos intentado reducirlas, excepto en las lecciones introductorias.

Se ha intentado que los ejemplos que aparecen en este libro no sean exclusivamente "científicos", con el objetivo de interesar a un mayor número de potenciales programadores; ello no debe hacer olvidar, sin embargo, que la

---

4 Consúltese la Bibliografía.

5 Véanse las referencias [Shaw 77] y [Lisk 77] para las sucesiones (allí llamados *iterators*) y las [Wirth 82] y [ADA* 83], así como el Rationale de ADA, para los módulos (*packages*).

6 Son lenguajes formales, pero no lenguajes en el sentido de idiomas, como parece que se pueda dar a entender.

programación es esencialmente una forma de manipulación simbólica, y que requiere, por tanto, de un cierto grado de formalismo, rigor y coherencia.[7]

Reconocimientos

Guillermo Alonso y Juan Carlos Paniagua leyeron meticulosamente el manuscrito, señalando errores e imprecisiones, y discutiendo con el autor explicaciones poco claras. Por tanto, son absolutamente responsables de los errores de este libro; yo, me lavo las manos.

Oriol Romeu ha estimulado a traves de discusiones largas, lúcidas, caóticas y creativas muchas de las reflexiones contenidas en el texto.

Elisabet Tubau, profesora del Curso de Programador 1984-85 y coautora de algunos trabajos relacionados con el lenguaje UBL, ha mantenido una estrecha relación con el proceso de diseño y modificación del lenguaje, aportando los resultados de sus experiencias.

José María Blasco
Barcelona, Octubre de 1985

---

7 Consúltese [Dijk 83] acerca de la relación entre la habilidad al programar y la facilidad comprensión matemática.

Notación empleada en este libro

Este texto es una recopilación de los apuntes distribuidos en el "Curso de Programador" antes citado efectuado en 1984-85, y es por lo tanto de lectura secuencial; se han añadido una lista de reglas sintácticas, así como numerosos indices, para facilitar su acceso como material de referencia.

Los primeros ordenadores disponian solo de impresoras con letras mayúsculas, razón por la cual muchos lenguajes se escriben tradicionalmente en mayúsculas; por otra parte, dado que el uso del ordenador facilita considerablemente la elaboración y mantenimiento de textos pero carecia hasta hace poco de simbolos especiales, se ha convertido en habitual en los libros de Informática renunciar a las notaciones usuales (como los subíndices o los operadores lógicos). En este texto[8] se ha intentado mantener la notación matemática clásica siempre que ha sido posible.[9] Se han escrito los programas y ejemplos en *cursiva*, con las palabras reservadas en **negrita**, siguiendo la convención de acentuar en los programas todo menos los identificadores reservados y predefinidos. Además, se han incluido en el texto algunas palabras en ***cursiva negrita***; esas palabras son definiciones, y se encuentran referenciadas en el indice alfabético.

---

8 Que ha sido elaborado mediante el complejo de composición de textos SCRIPT/VS de IBM, realizando algunas figuras con el paquete gráfico GDDM, y obteniendo los originales en una impresora de electroerosión IBM 4250.

9 En los apéndices al final del libro puede encontrarse una explicación sobre como utilizar las teclas del terminal para obtener efectos parecidos a algunos de los símbolos utilizados.

Cambios del lenguaje UBL respecto de la versión 0.1

La versión 0.2 del lenguaje UBL incorpora los siguientes cambios respecto de la anterior versión:

- Los identificadores reservados "**y**" ("**i**" en la versión catalana) y "**o**" se han substituido por los símbolos "∧" y "∨", respectivamente, dado que el hecho de que existiesen palabras reservadas de una sola letra causaba problemas en las declaraciones del estilo de

  var *x,y: real;*
  var *i,j,k: entero;*

  Igualmente, se ha substituido el identificador reservado "**no**" por el simbolo "~", y las formas lógicas "**y entonces**" y "**o sino**" por "∧∧" y "∨∨" respectivamente.

- El operador de concatenación "||" se ha substituido por "+".

- Se ha suprimido el concepto de **aplicacion**, dado que no se diferenciaba del de **tabla**.

- Las **secuencia**s de llaman ahora **sucesione**s.

- Las instrucciones **si, segun** y **decide** se han fundido en una única instrucción **si** (en su llamada ***forma general***), con el siguiente formato

  si
  □ *condicion* ⇒ *instruccion*
  ...
  □ *condicion* ⇒ *instruccion*
  □ **otros** ⇒ *instruccion*
  fin si;

  Lo que antes era la instrucción **segun** se presenta ahora como una abreviación de la forma general de **si**: por ejemplo, en

```
si
  □ c='A' ⇒ i1;
  □ c='B' ⇒ i2;
  □ c='C' ⇒ i3;
fin si;
```

se puede "sacar factor comun" la *c* para obtener la ***forma factorizada***:

```
si c es
  □ 'A' ⇒ i1;
  □ 'B' ⇒ i2;
  □ 'C' ⇒ i3;
fin si;
```

Finalmente, se conserva el **si-entonces-sino** clásico

```
si C entonces i1; sino i2; fin;
```

(con el nombre de ***forma simplificada***) y se presenta como abreviación de

```
si
  □ C      ⇒ i1;
  □ otros  ⇒ i2;
fin si;
```

Por último, en la forma factorizada se separan los valores por comas y no mediante el simbolo "|", como en la anterior versión.

La sintaxis de las **tupla**s con variantes se modifica consecuentemente.

- La instruccion **sal** pasa a escribirse **sal si** en vez de **sal cuando**.

Índice

Introducción 1
La multiplicación de Gitanos, versión 1 1
La multiplicación de Gitanos, versión 2 2
Lenguajes Naturales y Artificiales. Lenguajes de Programación 2
Métodos o algoritmos. Programas 3
Construcción de un programa para la multiplicación de Gitanos 3
Discusión 5
Ejercicios 6

Conceptos Básicos 7
Algoritmos y programas. 7
Compilación 7
Errores lógicos 8
Fases del diseño y elaboración de un programa 8
El modelo de ejecución secuencial 9
Objetos 10
Elementos del lenguaje UBL 10
Instrucción de asignación. Expresiones y evaluación 10
Iteración. Instrucción **repite** 11
Selección. Instrucción **si** 12
Obtención y presentación de datos: entrada y salida 12
Tipos: *caracter* y *entero* 13
Declaraciones. Declaraciones de objetos 13
Forma general de un programa en UBL 14
Un ejemplo: programa para contar las letras A 15
Ejercicios 16

Operadores lógicos 19
Condiciones complejas y operadores lógicos 19
Operadores relacionales. "Conjuntos de caracteres" 20
Ejemplo 1: Contar los dígrafos LA 21
Ejemplo 2: Contar los dígrafos LA interiores 23
Ejercicios 25

Sintaxis y estilo 27
Metalenguaje y metasímbolos. Reglas sintácticas 27
Un metalenguaje para la descripción sintáctica de UBL 28
Un ejemplo: Alfabeto del lenguaje UBL. 29
Símbolos 30
Comentarios 33
Escritura de programas 34

Convenciones de estilo 35
Ejercicios 39
Apéndice: tablas de identificadores reservados y predefinidos en UBL catalán y castellano. 40

Diseño descendente. Acciones y Condiciones **43**
Un ejemplo de diseño descendente 43
Otra solución al mismo problema. Forma general de la instrucción **si**. Instrucción **nada** 45
Acciones y Condiciones. Instrucciones **vale** y **acaba** 48
Notas 52
Ejercicios 52

Tiras de Caracteres **53**
Operaciones sobre tiras 54
Comparacion de tiras 55
Entrada y salida de tiras 56
Un ejemplo sencillo 56
Un ejemplo más complejo: apariciones de una palabra en un texto. 57
Ejercicios 61

Los tipos *real* y *logico*. Expresiones **63**
El tipo *real* 63
Precisión de los objetos de tipo *real* 63
Ejemplo: diferencias de cálculo en la serie armónica 64
El tipo *logico* 65
Operadores lógicos condicionales 66
Expresiones 67

Parámetros y declaraciones locales **71**
Entidades y declaraciones locales 71
Accesibilidad y existencia de las entidades locales 72
Reglas de reconocimiento de nombres 72
Parámetros 73
Efecto de Alias 75
Sintaxis completa de las declaraciones de subprograma y sus invocaciones 77

Funciones **79**
Necesidad de las Funciones. 79
Algunas funciones predefinidas; ejemplos simples 80
La declaración de tipo para *tiras* 81
Funcion *índice* 82
Cálculo de la raíz cuadrada de un número real mediante el método de bipartición . . 83
Ejercicios 85

Sucesiones **87**
Un ejemplo 87
Las **sucesiones** predefinidas *asc* y *desc* 88
La instrucción **para** 88
Existenciales 89
Otro ejemplo. Sintaxis de la instrucción **produce** 91
Un programa complejo: el justificador de textos 91
Ejercicio 96

Tipos Subrango y Enumerados **99**
Tipos enumerados 99
Ejemplos 101
Sintaxis 102
Los tipos *entero*, *caracter* y *logico* como enumerados 102
Subrangos 103
Cambio de fecha 104
Ejercicios 106

Otras instrucciones. Más sobre entrada y salida **107**
Instrucción **mientras** 107
Instrucciones **itera** y **sal** 108
Algunas equivalencias 110
Forma factorizada de la instrucción **si** 112
Formatos de entrada y salida 114
Entrada y salida interactiva 115
Cálculo del Máximo Común Divisor 116
Ejercicio 117

Conjuntos **119**
Los conjuntos como abreviación de expresiones 119
Tipos **conjunto** y operaciones 119
Algunos ejemplos 121

Tablas **125**
Familias indexadas de variables 125
Aplicaciones 126
Tabulación 127
Vectores 128
El tipo **tabla** 129
Operaciones 130
Variaciones sobre un mismo problema 131
Inversión de una serie de números 134
Matrices 135
Un ejemplo de tratamiento de matrices: generación de Cuadrados Mágicos 136
Generación de una tabla de números primos, con una discusión sobre optimización de programas 138
El problema de la siguiente permutación 140
Manipulación de Matrices 144
El problema de las ocho reinas 145
Algoritmos de ordenación 150
Búsqueda de elementos en una tabla 153
Ejercicios 154

Tuplas **157**
Productos Cartesianos 157
Uniones Disjuntas 157
Creación de un tipo para representar números complejos 158
Estructura de los objetos de tipo **tupla** 159
Tipo **tupla**. Nombres de campo 161
Tuplas con Variantes 162
Accesibilidad y existencia de los campos variantes 163
Las **tuplas** con variantes como uniones disjuntas 164
Instrucción **con** 165

Filas **167**
Acceso a los componentes de un fichero; estructura de **fila** 167
Operaciones 169
Convenciones de Fin de Fila 171
Fusión de ficheros 172
Filas de caracteres. El tipo predefinido *texto* 172
Control de Stocks 175

Subprogramas como parámetros **179**
Integración por trapecios 180
Filtros de sucesiones 182

Recursividad **183**
Introducción 183
Recursión directa e indirecta. Definiciones e implementaciones de subprogramas 185
La sucesión de Fibonacci 185
Las torres de Hanoi 187
Descomposición de un entero en suma de dados 190
Las Ocho Reinas, en versión recursiva 192
Ejercicios 194

Nombres **195**
Principal uso de los objetos de tipo nombre 197
Listas unidireccionales 199
Arboles 201
Ejercicios 206

Módulos **207**
Dos implementaciones de un mismo problema 210
Acceso controlado a variables 218
Pilas 219

Apéndice 1: Lista de reglas sintácticas **223**

Apéndice 2: El compilador UBL/CMS versión 0.2 ..... **231**
Estructura general 231
Representación interna de los datos 232
Representación de los valores de tipo *logico* 232
Representación de los valores de tipo *caracter* 232
Representacion de los valores de tipo *entero* o enumerado 232
Representación de los valores de tipo *real* 233
Representación de los valores de tipo **nombre** 234
Representación de los valores de tipo **conjunto** 234
Representación de objetos estructurados 234
Instrucciones '%' 235
Utilización: instrucciones de CMS 236
Compilación 236
Ejecución 237
Escritura de programas en terminales de tipo 3278 238

Bibliografia **239**

Índice Alfabético **241**

Lista de Figuras

Figura 1. Método de la multiplicación de Gitanos 1
Figura 2. Sintaxis de la instrucción de asignación 10
Figura 3. Sintaxis de la instrucción **repite** 11
Figura 4. Sintaxis de la instrucción **si** 12
Figura 5. Declaración de **variables** 13
Figura 6. Declaración de **constantes** 14
Figura 7. Sintaxis de un **programa** 14
Figura 8. Tablas de verdad para los operadores lógicos 19
Figura 9. Operadores relacionales 20
Figura 10. Alfabeto del lenguaje UBL 30
Figura 11. Vocabulario del lenguaje UBL 31
Figura 12. Sintaxis y estilos de escritura para la forma simple de la instrucción **si** 37
Figura 13. Sintaxis de la instrucción de asignación 38
Figura 14. Sintaxis y estilos de escritura para la instrucción **repite** 38
Figura 15. Sintaxis y estilo de escritura para un **programa** 38
Figura 16. Sintaxis y estilos de escritura para las declaraciones de objeto 39
Figura 17. Tabla de identificadores reservados en UBL Castellano 40
Figura 18. Tabla de identificadores predefinidos en UBL Castellano. 40
Figura 19. Tabla de identificadores reservados en UBL Catalán 41
Figura 20. Tabla de identificadores predefinidos en UBL Catalán. 41
Figura 21. Sintaxis y estilo de la forma general de la instrucción **si** 45
Figura 22. Sintaxis de la instrucción **nada** 46
Figura 23. Sintaxis de la instrucción **vale** en condiciones 49
Figura 24. Sintaxis y estilo para declaraciones de **acciones** (Simplificado) 51
Figura 25. Sintaxis de la instrucción de invocación (Simplificada) 51
Figura 26. Sintaxis y estilo para declaraciones de **condicion** (Simplificado) 51
Figura 27. Sintaxis de la instrucción **acaba** 52
Figura 28. Operadores lógicos condicionales 66
Figura 29. Sintaxis de una expresión 68
Figura 30. Sintaxis de la lista de parámetros 74
Figura 31. Sintaxis de las declaraciones de subprograma 77
Figura 32. Sintaxis de la lista de argumentos 77
Figura 33. Sintaxis de la instrucción de invocación a una **accion** 77
Figura 34. Sintaxis de una invocación de **condicion** 78
Figura 35. Sintaxis de una invocación de **funcion** 80
Figura 36. Sintaxis de la instrucción **vale** 80
Figura 37. Sintaxis de una declaración de tipo *tira* 81
Figura 38. Fragmento del cálculo de la raíz de 2 por bipartición 84
Figura 39. Sintaxis y estilo de la instrucción **para** 89
Figura 40. Sintaxis de un existencial 90
Figura 41. Sintaxis de la instrucción **produce** 91

Figura 42. Sintaxis de los tipos y sus declaraciones 102
Figura 43. Sintaxis de los tipos enumerados. 102
Figura 44. Sintaxis de un tipo subrango 103
Figura 45. Instrucciones 107
Figura 46. Sintaxis y estilo de la instrucción **mientras** 108
Figura 47. Sintaxis y estilo de la instrucción **itera** 108
Figura 48. Sintaxis de la instrucción **sal** 109
Figura 49. Esquema normal de utilización de la instrucción **itera** 109
Figura 50. Sintaxis y estilo de la forma factorizada de la instrucción **si** 112
Figura 51. Esquema general de diálogo interactivo 116
Figura 52. Sintaxis de un conjunto 119
Figura 53. Sintaxis de un tipo **conjunto** 120
Figura 54. Una **tabla** de 7 *enteros*, contemplada como **tabla** y como familia de variables subindicadas 126
Figura 55. Fragmento de una aplicación de los caracteres en los caracteres representada mediante una **tabla** 127
Figura 56. Tabla de los horarios de salida del trabajo en días laborables 128
Figura 57. Sintaxis de un tipo **tabla** 129
Figura 58. Sintaxis de una variable con sus posibles selectores 130
Figura 59. Utilización combinada de una **tabla** y un apuntador 134
Figura 60. Una matriz con una submatriz 136
Figura 61. Ejemplo de Cuadrado Mágico de lado 5 136
Figura 62. Una de las soluciones al problema de las Ocho Reinas 146
Figura 63. Fragmento del proceso de Backtracking para las Ocho Reinas en un tablero de 4x4 148
Figura 64. Ordenación por el método de la burbuja 152
Figura 65. Declaraciones y operaciones sobre un tipo *fecha* 160
Figura 66. Sintaxis y estilo de un tipo **tupla** 161
Figura 67. Sintaxis de una **tupla** con variantes 163
Figura 68. Declaraciones y gráficos para un tipo **tupla** con variantes 164
Figura 69. Sintaxis y estilo de la instrucción **con** 165
Figura 70. Un fichero de *enteros* 167
Figura 71. El mismo fichero, su **fila** y la ventana 168
Figura 72. Sintaxis de un tipo **fila** 168
Figura 73. Un fichero de texto 173
Figura 74. Sintaxis de un parámetro por subprograma 179
Figura 75. Ejemplo de integracion por trapecios. 181
Figura 76. Arbol de invocaciones para Fibonacci(4) 186
Figura 77. Las Torres de Hanoi: posición inicial con 5 discos 188
Figura 78. Movimientos para el caso de dos discos 189
Figura 79. Grafo de invocaciones para Hanoi A,B,C,2 190
Figura 80. Descomposiciones posibles del número 7 191
Figura 81. Un objeto de tipo **nombre** y el objeto nombrado 195
Figura 82. Sintaxis de una declaración de tipo **nombre** 195
Figura 83. Una lista unidireccional de *enteros* 199
Figura 84. Inserción en una lista unidireccional 200
Figura 85. Supresión del siguiente de un elemento en una lista unidireccional 200
Figura 86. Representación de un árbol binario 202
Figura 87. Un árbol y sus recorridos 204
Figura 88. Sintaxis de las declaraciones de **modulo** 207
Figura 89. Sintaxis de la declaración **usa** 208
Figura 90. Tabla de palabras y sus frecuencias de aparición 211
Figura 91. Una pila de *enteros* 220
Figura 92. Estructura y funcionamiento del compilador UBL/CMS 231

Figura 93. Formato de la instrucción UBL 236
Figura 94. Escritura de símbolos que no están en las terminales 238

Lista de Algoritmos

Algoritmo 1. Multiplicación de Gitanos. 5
Algoritmo 2. Fases del diseño y elaboración de un programa 9
Algoritmo 3. Contar las As de una frase terminada por un punto 16
Algoritmo 4. Contar los dígrafos LA. 23
Algoritmo 5. Contar los dígrafos LA interiores. 24
Algoritmo 6. Ejemplo de programa ilegible 35
Algoritmo 7. Media de las As por frase en un texto. 48
Algoritmo 8. Contar la media de As por frase en un texto (con acciones y condiciones) 50
Algoritmo 9. Descomposición en Nombre y Apellidos 57
Algoritmo 10. Apariciones de una palabra en un texto 60
Algoritmo 11. Calculo de la serie armónica 65
Algoritmo 12. *Indice* para tiras de caracteres 82
Algoritmo 13. Raíz cuadrada por Bipartición 84
Algoritmo 14. Para saber si un entero es primo 90
Algoritmo 15. Justificador de textos 96
Algoritmo 16. Halla la fecha siguiente a una dada. 105
Algoritmo 17. Cálculo del Máximo Común Divisor 117
Algoritmo 18. Generación de números primos la Criba de Eratóstenes 124
Algoritmo 19. Filtra entre 10 números los mayores que su media 132
Algoritmo 20. Filtra entre una serie de números terminada por 0 los mayores que su media 133
Algoritmo 21. Filtra entre una serie de C números los mayores que su media 133
Algoritmo 22. Invierte una serie de números 135
Algoritmo 23. Generación de Cuadrados Mágicos. 138
Algoritmo 24. Generación de números primos 140
Algoritmo 25. Generación de permutaciones de N elementos por el método de la Siguiente Permutación 143
Algoritmo 26. Manejo de matrices 145
Algoritmo 27. Las Ocho Reinas 150
Algoritmo 28. Ordenación de vectores, método elemental 151
Algoritmo 29. Ordenación de vectores por el método de la burbuja 153
Algoritmo 30. Búsqueda binaria 154
Algoritmo 31. Suma de números complejos 159
Algoritmo 32. Calcula la media de una fila de enteros 171
Algoritmo 33. Fusión de ficheros 172
Algoritmo 34. Copia la fila de texto predefinida *entrada* en la fila de texto predefinida *salida* 175
Algoritmo 35. Control de Stocks 177
Algoritmo 36. Integración numérica por el método de los trapecios 181
Algoritmo 37. Un filtro para sucesiones de enteros 182

Algoritmo 38. La sucesión de Fibonacci 186
Algoritmo 39. La sucesión de Fibonacci, versión iterativa 187
Algoritmo 40. Las Torres de Hanoi 188
Algoritmo 41. Descomposición en suma de dados 192
Algoritmo 42. Las Ocho Reinas en versión recursiva 193
Algoritmo 43. Sucesiones para el tratamiento de listas unidireccionales 201
Algoritmo 44. Funciones para el manejo de árboles 203
Algoritmo 45. Inserción en un árbol binario 203
Algoritmo 46. Sucesiones *inorden*, *preorden* y *postreorden* sobre un árbol 205
Algoritmo 47. Subprogramas matemáticos 210
Algoritmo 48. (Incompleto) Frecuencia de aparición de palabras en una frase 213
Algoritmo 49. Tabla de frecuencias mediante **nombres** 215
Algoritmo 50. Tabla de frecuencias mediante **tablas** 217
Algoritmo 51. Acceso controlado a una variable 219
Algoritmo 52. Definición de una *pila* de enteros 220
Algoritmo 53. Implementación de una *pila* de enteros mediante **nombres** 221
Algoritmo 54. Implementación de una *pila* de enteros mediante **tablas** 222

Introducción

La multiplicación de Gitanos, versión 1

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

Ilustraremos el método 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. Método de la multiplicación de Gitanos

Supongamos que queremos multiplicar 23 y 45. Primero los ponemos uno al lado del otro (1). En la línea 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). Después tachamos las filas que tengan un número par en la segunda columna (4), y sumamos los números que quedan en la primera columna (5). El resultado de la suma es el producto deseado (el método puede aplicarse a cualquier otro par de números naturales).

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

La multiplicación de Gitanos, versión 2

Otra versión del mismo método 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 Programación

Estas explicaciones pueden ser suficientes para que una persona comprenda el método 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 tecnología, no son comprensibles directamente por un ordenador. Es sabido que el lenguaje escrito (y también el hablado) suele ser fuente de ambigüedades (la frase "Adiós" puede ser una despedida o una invitación a marcharse, según el tono y la circunstancia). Los ordenadores no pueden decidir de qué se está hablando como lo hace una persona.

Así, suelen utilizarse en la mayoría de los ordenadores lenguajes artificiales o de ***Programación***[1] (que llamaremos aquí simplemente ***lenguajes***), más sencillos y menos ambiguos en su estructura que el lenguaje hablado. En la mayoría de los casos, suelen ser lenguajes mixtos entre la notación matemática y el lenguaje habitual. Las hipótesis implícitas al utilizar estos lenguajes varían con cada uno de ellos, así como el modelo conceptual sobre el que se construyen las instrucciones al ordenador. Aquí utilizaremos el Lenguaje de la Universidad de Barcelona (UBL), que permite escribir programas en una notación más próxima al Castellano (o Catalán) que otros lenguajes.

---

[1] Recordamos que, en todo el texto, las palabras que aparecen en negrita y cursiva son definiciones, de las cuales puede encontrarse una referencia en el índice que se encuentra al final del libro.

Métodos o algoritmos. Programas

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

En el ejemplo mostrado, deseamos proporcionar al ordenador información sobre como realizar la multiplicación de Gitanos. Queremos que lo aprenda *en general*, esto es, que sea capaz de aplicarlo para multiplicar cualquier par de números, y no sólo 23 y 45. Para ello, deberemos construir un programa (una descripcion del método), de acuerdo a las convenciones del lenguaje utilizado.

Desarrollamos a continuación un programa en el lenguaje UBL para describir la versión 2 de la multiplicación de Gitanos; el propósito de esta exposición es proporcionar una impresión general de cómo se trabaja con el lenguaje: muchos conceptos se entenderán sólo aproximadamente, y algunos detalles sintácticos (como la presencia de puntos y coma o la palabra **fin**[2]) podrán parecer arbitrarios. Todo ello se explicará detalladamente en capítulos posteriores.

Construcción de un programa para la multiplicación 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 línea a partir de la anterior pueden expresarse precisamente mediante las ***instrucciones***:

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 escribirá

---

2 Las palabras que aparecen en negrita son elementos privilegiados del lenguaje, llamados ***palabras reservadas***, que se escriben resaltados para clarificar la estructura del programa. Se estudiarán más adelante, con los demás símbolos que integran la notación.

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

La notación utilizada requiere cierta explicación: *2 * m* significa "el producto de 2 por *m*" (el símbolo "*" substituye a las aspas de multiplicar en la mayoría de ordenadores); "**div**" significa "dividido por"; en cuanto al símbolo "←", que se lee "toma por valor", es el modo de dar valor a un identificador. A la operación de dar valor la llamaremos ***asignación***.

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 expresará en UBL del siguiente modo

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

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

m	*n*	*p*
23	*45*	*23*

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

m	*n*	*p*
46	*22*	*23*

y así sucesivamente. Es necesario informar al ordenador de cuántas variables necesitamos, qué nombres tienen y qué tipo de información van a contener: esto se hace mediante la ***declaración***

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

Inicialmente (al empezar el proceso), las variables deberán 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* según *n* sea impar). Hemos dicho que proporcionamos información a la máquina sobre el método general; cada utilización (o ***ejecución***) del método deberá hacerse con un par concreto de valores iniciales: el ordenador "pedirá" estos valores (por la pantalla u otro medio) mediante una operación que llamaremos ***lectura***:

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

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

```
escribe p;
```

Así, el programa completo se escribirá:

```
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. Multiplicación de Gitanos.

La diferente posición (o ***indentación***[3]) de las líneas con respecto al margen izquierdo se utiliza para señalar la subordinación lógica de los instrucciones empleadas: "*m* ← *2 * m*" es una de las instrucciones que deben repetirse, y por ello se encuentra más a la derecha que la palabra **repite**.

Discusión

Se han presentado los conceptos principales utilizados en la mayoría de lenguajes de programación: las ***variables*** como pizarras o cajas que pueden contener información de determinado tipo, y que deben definirse o ***declararse***; la ***asignación***, que permite alterar el valor de estas variables; la ***selección*** o ***toma de decisión*** (instrucción **si**), que permite que el ordenador obedezca una instrucción u otra

3 La palabra correcta sería ***sangrado***, para expresar el desplazamiento horizontal de una línea o grupo de líneas respecto del margen. ***Indentación*** en un anglicismo extendido entre la jerga informática.

según se cumpla o no determinada condición; y la ***iteración*** (instrucción **repite**), que permite realizar repetidamente un grupo de instrucciones.

Hemos elegido la versión 2 del método para escribir el programa, ya que la versión 1, al requerir volver sobre los resultados anteriores (p. ej., tachando columnas) para obtener el resultado, es difícil 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 líneas que se producirán, y no hay manera, sin tener una variable para cada número, de "recordar" los valores de estas variables). Esto es frecuente en programación: un método puede ser adecuado para ser realizado a mano, pero una versión aparentemente más complicada puede ser más útil o simple en un ordenador.

Ejercicios

1. Puede ser interesante que el programa proporcione alguna información sobre lo que se desea, y también que, además del resultado final, escriba todos los pasos del cálculo. Sabiendo que es posible utilizar la instrucción *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, en esencia, el método se reduce a una multiplicación simple en base 2.

Conceptos Básicos

Algoritmos y programas.

Un ***algoritmo*** es un método para realizar algún tipo de proceso (de un modo más técnico, es una clase de trasformaciones de datos); un ***programa*** es la especificación de un algoritmo en determinado lenguaje. Los programas suelen escribirse con el propósito de que sean ***interpretados*** o ***ejecutados*** por un ordenador; esta interpretación consiste en la ejecución de las ***instrucciones*** que el programa especifica.

Pocos programas se escriben para realizar un proceso concreto; la mayoría de ellos aceptan ***datos***, que actúan como parámetros del proceso. Podemos verificar que un programa "funciona", esto es, que sus especificaciones responden a nuestro propósito al escribirlo, para unos datos concretos; verificarlo para todos los datos posibles es, en general, una tarea inviable, de modo que la ***corrección*** de un programa deberá establecerse a partir de un diseño y análisis cuidadosos del mismo.

Compilación

Un programa se escribe, y se suministra al ordenador, en forma de texto. El ordenador verifica que el texto esté escrito conforme a las reglas que prescribe el lenguaje, detectando construcciones erróneas. Al mismo tiempo, "comprende" el programa, y posiblemente lo traduzca a un formato interno que facilita su posterior ***interpretación***[4]. A este doble proceso lo llamamos ***compilación***, y ***compilador*** al agente que lo realiza. La comprensión del texto (por el ordenador) sólo es posible si éste no contiene ***errores de compilación*** (en cuanto a las reglas de escritura del lenguaje); en caso de que los haya, el compilador informa de ellos mediante algún tipo de mensaje: la mayoría de compiladores producen un ***listado***, que suele contener, además del programa (posiblemente "embellecido" con algún mecanismo, o incluso reescrito para aumentar su legibilidad), información adicional

---

4 Tradicionalmente se distingue entre ***compiladores*** e ***intérpretes***, haciendo referencia a si el texto del programa ha sido convertido a una secuencia equivalente de instrucciones máquina, o se conserva y se ejecuta a partir de un formato (parecido al) original. En todo caso, se haya convertido o no el programa, se precisa un agente que lo ejecute: a este agente es al que nosotros llamamos ***intérprete***.

(numeración de las instrucciones, estadísticas de compilación, errores detectados, etc).

Errores lógicos

Hay que notar que una compilación 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 sintácticamente 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 lógico***; en este caso, no hay ningún mecanismo que nos informe de las causas del error (ya que no hay nadie, aparte de nosotros mismos, que pueda saber qué intentábamos decir al escribir un programa). La corrección de los errores lógicos es generalmente mucho más complicada que la de los de compilación, y a veces requiere de mecanismos auxiliares. Al conjunto de acciones que la constituyen se le suele llamar ***depuración*** de un programa.

Fases del diseño y elaboración de un programa

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

- ***Análisis y delimitación teóricos*** del problema, que deberá realizarse sobre bases lo más 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 comentarán en capítulos posteriores.
- ***Sometimiento del programa escrito al compilador***. Si se detectan errores en esta fase, será necesario corregirlos y repetir la compilación.
- ***Prueba*** del programa con una muestra de datos razonablemente amplia. Si se producen fallos, será necesario revisar el programa y volver a compilar, o incluso revisar el modelo teórico. ***Depuración*** en este caso para detectar con exactitud las partes incorrectas del programa.

que también pueden presentarse en forma de algoritmo en UBL[5]:

---

5 El signo "~" corresponde al "no" lógico.

```
diseño_teorico;
escritura_del_programa;
compilación;
si hay_errores entonces
  repite
    corrección_de_los_errores;
    compilación;
  hastaque ~ hay_errores;
fin si;
prueba;
si ~ funciona entonces
  repite
    depuración_e_identificación_de_los_errores;
    corrección_de_los_errores;
    compilación;
    si hay_errores entonces
      repite
        corrección_de_los_errores;
        compilación;
      hastaque ~ hay_errores;
    fin si;
    prueba;
  hastaque funciona;
fin si;
```

Algoritmo 2. Fases del diseño y elaboración de un programa

El modelo de ejecución secuencial

Supondremos que cada programa define un ***orden*** estricto en la ejecución de las instrucciones que componen el programa; una instrucción se obedece, efectua o ***ejecuta*** realizando la ***acción*** que designa; tal ejecución tiene un ***efecto*** sobre el ***estado*** del programa[6], efecto que podemos tomar en consideración al ejecutar la siguiente instrucción. Dicho más sencillo, cada instrucción se ejecuta completamente antes de ejecutar la siguiente. Nótese que el orden de ejecución de las instrucciones no tiene porque corresponder con el orden textual de éstas: por ejemplo, una instrucción repetitiva determina la ejecución reiterada del mismo grupo de instrucciones.

6 O ***estado de la ejecución del programa***: probablemente sean afectados los valores de las variables.

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 página 5, *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 información de determinado ***tipo***, y posiblemente contiene un ***valor***:

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

Hay dos operaciones básicas aplicables a los objetos, que corresponden a las ideas intuitivas de consultar y alterar el valor escrito en una pizarra: las llamaremos respectivamente ***inspección*** y ***modificación***. Convendremos en que [el valor de] cualquier objeto podrá ser inspeccionado, pero no todos los objetos podrán 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 ***constantes literales*** o simplemente ***literales***.

Elementos del lenguaje UBL

Un programa se compone de ***declaraciones*** de objetos (y otras entidades que se estudiarán más adelante) e ***instrucciones*** que operan a partir de y modifican el valor de esos objetos. A continuación se describen algunas de las instrucciones básicas.

Instrucción de asignación. Expresiones y evaluación

La instrucción más sencilla es la de ***asignación***:

```
variable ← expresion;
```

Figura 2. Sintaxis de la instrucción de asignación

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

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

v ← *3;*

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

Entendemos por ***expresión*** cualquier fórmula, escrita siguiendo las convenciones del lenguaje, que serán descritas más adelante. Por el momento, bastará saber que una expresión puede contener tanto literales como otros objetos, tomándose éstos últimos por una especificación de su valor; podrán realizarse las operaciones aritméticas usuales, utilizando paréntesis cuando sea necesario. Al manejar números enteros, se dispondrá de los operadores de adición (+), substracción (-), multiplicación (*), división o cociente (**div**) y módulo (**mod**). Nos referiremos al cálculo del valor de una expresión hablando de su ***evaluación***.

Iteración. Instrucción repite

En la mayoría de los lenguajes ***imperativos*** (es decir, que utilizan instrucciones) existen instrucciones ***iterativas*** o repetitivas, que se utilizan para efectuar la ejecución repetida de otra instrucción. La idea es que un proceso puede hacerse por partes, realizando en cada repetición una parte del proceso total. Así pues, es necesario que cada ejecución de la instrucción repetida nos "acerque" en cierto sentido a la consecución del objetivo propuesto.

Se ha visto un ejemplo de instrucción iterativa (**repite**) en el Algoritmo 1 en la página 5. La forma general de la instrucción es

```
repite
  instruccion(es)
hastaque condicion;
```

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

donde ***condición*** significa cualquier expresión que tenga un valor que sea *cierto* o *falso*, p. ej., una comparación del estilo de *n* = *1*.

El efecto de la ejecución de esta instrucción, de acuerdo con su significado intuitivo, consiste en ejecutar la(s) instruccion(es) subordinadas, evaluar la condición, 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 condición se cumple a mitad de la ejecucion de la instrucción subordinada su ejecución se detenga: en

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

aunque la ejecución de *a*←*2* hace que la condición *a=2* se cumpla, ésta no se evalua hasta que finaliza la ejecución de *b*←*b-1*.

Selección. Instrucción si

Otro tipo de instrucción universalmente utilizado es la ***selección*** o ***toma de decisión***, que supone en el intérprete del lenguaje la facultad de ejecutar una instrucción u otra según se verifique determinada condición. El ejemplo mas sencillo de instrucción de selección es la instrucción **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 instrucción* si**

cuyo efecto consiste en evaluar la condición, y ejecutar la(s) instruccion(es)$_1$ o la(s) instruccion(es)$_2$ según 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 sólo se haga en instrucciones textualmente muy cortas).

Obtención y presentación 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 tradición ***instrucciones de entrada y salida***) para realizar estas operaciones. Así, la instrucción

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 aparición de un número entero en el medio externo, posiblemente precedido de cualquier número de espacios en blanco; la lectura de caracteres se realiza carácter por carácter, contando el espacio en blanco como un carácter como los demas.

Por su parte, la instrucción

escribe expresiones;

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

Tipos: *caracter* y *entero*

El lenguaje define varios ***tipos*** de datos, además de la posibilidad de crear nuevos tipos (como se verá en capitulos posteriores). Se han tratado ya objetos de tipo *entero*; definiremos ahora un nuevo tipo, que llamaremos ***caracter***: un objeto de tipo *caracter* podrá tener como valor cualquier letra (mayúscula o minúscula), número o signo de puntuación (incluyendo el espacio en blanco, paréntesis, llaves, corchetes y posiblemente otros, dependiendo del ordenador). Los literales de tipo *caracter* se escriben entre apóstrofes: *'A'*, *'a'*, *'2'*, *' '* o *'%'* son caracteres. Se denota por *''''* el carácter cuyo valor es el apóstrofe.

Declaraciones. Declaraciones de objetos

Las entidades que se manipulan en un programa (objetos u otros tipos de entidad que se estudiarán) 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 declaración pueda ser detectado por el compilador.

Las variables se declaran especificando su tipo:

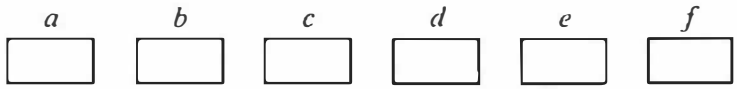
var *identificador(es) : tipo;*

Figura 5. Declaración de variables

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

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 declaración es necesaria para que el intérprete "cree" las variables y podamos trabajar con ellas:



Una declaración 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 ***inicialización*** [ver más adelante]).

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

const *identificador = valor;*

***Figura 6. Declaración de* constantes**

donde *valor* puede ser un literal, otra constante o una expresión que involucre solo constantes:

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

define dos objetos constantes de nombres *max* y *max2*, tipo *entero* (implícito 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 notará que se separan (conceptual y textualmente) las declaraciones de las instrucciones mediante la palabra **haz**.

Un ejemplo: programa para contar las letras A

Definición 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 sólo las As mayúsculas, y supondremos que la frase contiene al menos una letra.

Análisis de una posible solución: 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 número de sus apariciones.

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

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

Obtendremos los caracteres de la frase mediante la ejecución repetida de

```
lee c;
```

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

```
n ← n + 1;
```

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

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

Obviamente, será necesario indicar que, antes de examinar cualquier carácter, el número de apariciones de la letra A es cero:

```
n ← 0;
```

Si añadimos instrucciones para informar de qué proceso realiza el programa y de cuál 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

Discusión: la iteración es correcta: en primer lugar, nos aproximamos al fin del proceso (ya que la instrucción *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 condición de terminación $c='.'$.

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

Ejercicios

1. Algunos lenguajes definen una instrucción de ***asignación múltiple*** que tiene como efecto, en este orden, la evaluación de las expresiones de la parte derecha de la asignación y la asignación de los valores resultantes a las correspondientes variables de la parte izquierda:

   $x, y \leftarrow 2, 3;$

   tiene el mismo efecto que

   $x \leftarrow 2; y \leftarrow 3;$

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

   $x, y \leftarrow y, x;$

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

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

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

Operadores lógicos

Condiciones complejas y operadores lógicos

Recordaremos que hemos llamado ***condiciones*** a las expresiones que pueden ser ciertas o falsas (Como se verá, estas expresiones tienen su tipo: el tipo *logico*, que será estudiado más adelante). En muchas ocasiones es necesario utilizar varias condiciones a la vez, ya sea en forma de conjunción, disyunción o negación. Definiremos tres operadores "∨" (se lee *o*), "∧" (se lee *y*) y "~" (se lee ***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 lógica tradicional:

$A \vee B$	es *cierto*	si A o B son *ciertos*
$A \wedge B$	es *cierto*	solo si A y B son *cierto*
$\sim A$	es *cierto*	si A es *falso*

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

A	B	$A \wedge B$	$A \vee B$	$\sim A$
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 lógicos

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

```
(a = 3) ∧ (b = 5)
(c = 'a') ∨ (c = 'A')
~ (n = 3 ∧ m = -3)
```

y "funcionan" conforme a su significado intuitivo.

Tanto el operador ∧ como el ∨ son asociativos. Puede escribirse

$A \vee B \vee C$

para expresar cualquiera de las condiciones

$(A \vee B) \vee C$
$A \vee (B \vee C)$

y similarmente para el operador ∧. También son conmutativos (esto es, $A \vee B$ y $B \vee A$ son equivalentes, y similarmente para ∧), pero no pueden mezclarse directamente: aunque la Matemática define una prioridad entre estos operadores, poca gente la conoce, y cuesta saber si

$A \wedge B \vee C$ (1)

significa

$A \wedge (B \vee C)$ (2)
o
$(A \wedge B) \vee C$ (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 paréntesis.

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 interpretación, aplicada a valores de tipo *entero*, es la intuitiva; en el caso de los caracteres, por el contrario, no hay ninguna ordenación "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 heterogéneos entre sí, algunos "naturalmente ordenados", como las

letras y los números, y otros no, como los caracteres especiales. De todos modos, subsisten algunas preguntas, p. ej., "¿qué sentido tiene decir que *'a'* es mayor, menor o igual que *'A'*?". Aunque la estructura natural de este conjunto sería el de un orden parcial, normalmente cada ordenador define un orden total sobre los caracteres más o menos arbitrario, (añadiendo estructura a su conjunto; esto quiere decir que los ordena todos: podrá compararse cualquier par de caracteres), que suele conocerse como ***conjunto de caracteres*** [del inglés "character set"]. Estos conjuntos de caracteres suelen variar entre ordenadores, y aún entre distintos modelos del mismo fabricante; algunos son más amplios que otros (por ejemplo, los hay que no incorporan las letras minúsculas). Las suposiciones básicas que cabe esperar que satisfaga todo conjunto de caracteres son las siguientes:

1. El espacio en blanco (' ') es *menor* que cualquier otro carácter "imprimible".[7]
2. Las letras están ***bien ordenadas*** entre sí, esto es, *'a' < 'b' < 'c' < ... < 'z'*, y *'A' < 'B' < 'C' < ... < 'Z'*.[8]
3. Los números tambien estan bien ordenados (*'0' < '1'*, etc.), y además son ***contiguos***: no existe ningun caracter entre el '0' y el '1', ni entre el '1' y el '2', etc. La propiedad de contigüidad, deseable también para las letras, no se cumple para los conjuntos de caracteres de muchos ordenadores.

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

Ejemplo 1: Contar los dígrafos LA

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

Análisis de una posible solución: El problema es similar en su estructura al resuelto en el Algoritmo 3 en la página 16. Podremos, pues, utilizar el mismo esquema, aunque en este caso la unidad de tratamiento no será el caracter, sino el par de caracteres o *dígrafo*.

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

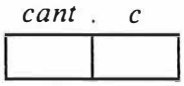
---

[7] La mayoría de *conjuntos de caracteres* constan de un número de caracteres "imprimibles" y de otros, llamados ***caracteres de control***, que no lo son y suelen utilizarse para funciones internas de los dispositivos de representación de textos, como saltos de página o cambios de línea.

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

var *cant, c: caracter;*

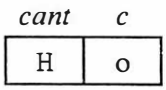
donde *c* representa el último carácter que analizamos y *cant* el carácter anterior a éste; y la pareja (*cant,c*), el par de caracteres que estamos analizando:



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

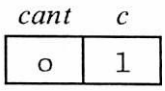
Si la frase es: `'Hola.'`, las parejas serán `'Ho'`, `'ol'`, `'la'` y `'a.'`.

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



Antes de cambiar de pareja

al pasar al siguiente par de caracteres,



Después de cambiar de pareja

cant toma por valor el anterior valor de *c*
c es el carácter recién leido

Así, escribiremos

cant ← *c; lee c;*

para realizar el paso de un par al siguiente.

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

$(cant = 'L' \vee cant = 'l') \wedge (c = 'A' \vee c = 'a')$

Por último, hay que tener en cuenta que, si deseamos realizar un tratamiento sistemático de todos los casos posibles, hemos de prever la posibilidad de que sólo haya una letra en la frase. Esto puede complicar el programa (habría que mirar los dígrafos sólo si la frase tuviera más de una letra), de modo que propondremos una solución alternativa: considerar que la frase esta "extendida" con un espacio en blanco a la izquierda (puede verse fácilmente que esta transformación no altera el

resultado del programa). Para ello, inicializaremos $c \leftarrow ' '$ con lo que obtendremos la siguiente versión 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' ∨ cant = 'l') ∧ (c = 'A' ∨ 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 dígrafos LA interiores

Una modificación del programa anterior consiste en contar sólo los digrafos LA *que se encuentren a mitad de palabra*, esto es, no los que empiezan o terminan una. En este caso, habrá que inspeccionar cuaternas (o cuadruplas; o grupos de cuatro) de caracteres: los situados en el medio habrán 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 convención 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 bastará con asegurarnos de que *c3* sea un espacio en blanco. Saber si estamos a final de palabra requiere, además de verificar que *c* no sea un blanco, verificar que no sea un punto, ya que podría ser que la última palabra de la frase terminase con el digrafo LA seguido inmediatamente de un punto; y, en este caso, no debería ser contado.

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

Ofrecemos una versión 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 ~ (c3 = ' ') ∧ ~ (c = ' ' ∨ c = '.') ∧
      (c2 = 'L' ∨ c2 = 'l') ∧ (c1 = 'A' ∨ 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 dígrafos LA interiores.

Se notara que las (sub)expresiones negadas, como ~ *(c3=' ')*, hubiesen podido ser escritas mas sencillamente como *c3≠' '*, utilizando el operador de desigualdad "≠". Se ha utilizado ~ para mostrar su funcionamiento.

Ejercicios

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

   $\sim (A \wedge B) = \sim A \vee \sim B$
   $\sim (A \vee B) = \sim A \wedge \sim B$

2. Simplificar las siguientes expresiones lógicas:

   $\sim \sim A$
   $\sim (A \wedge B \wedge C)$
   $(A \wedge B) \vee (A \wedge \sim B)$
   $(A \vee B) \wedge (A \vee \sim B)$

3. Ver qué cambios son necesarios en el Algoritmo 3 en la página 16 para que cuente las As mayúsculas y minúsculas.

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

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

Sintaxis y estilo

Definiremos un formalismo que nos permitirá describir con total precisión la sintaxis de las construcciones del lenguaje UBL. Esto es necesario (ya que las descripciones informales, utilizadas hasta ahora, son inadecuadas para describir con precisión lenguajes de esta naturaleza), y desde luego conveniente (nos permite disponer de un criterio fiable para decidir sin ambigüedad cuáles son las construcciones correctas en el lenguaje). Igualmente, definiremos convenciones estilísticas (diferentes de la sintaxis) que, sin ser obligatorias, permitirán, de ser seguidas, el intercambio y la comprensión fáciles de los programas.

Metalenguaje y metasímbolos. Reglas sintácticas

Todo lenguaje define reglas para la formación 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 análisis hace tiempo que no se aplica, vamos a suponerlo así para simplificar la argumentación). No hay ninguna dificultad en aceptar que esto puede expresarse mediante la siguiente formalización:

Frase = Sujeto Predicado
Predicado = Verbo Complemento

Estrictamente, las dos líneas anteriores constituyen de por sí "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 sintácticas*** a las frases del metalenguaje.

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

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

Sujeto = Yo | Tu

Sin embargo, caemos aquí en cierta confusión, ya que las palabras "Sujeto" y "Yo" o "Tu" pertenecen a categorias distintas. En efecto, pues "Sujeto" representa a cualquier sujeto, es una construcción genérica perteneciente al metalenguaje, y en cambio "Yo" o "Tu" son palabras concretas del lenguaje. Llamaremos símbolos ***no terminales*** a los que denotan a las construcciones genéricas, y ***terminales*** a los elementos del lenguaje. Utilizaremos algún tipo de distinción gráfica (que se especificará más adelante) para determinar si un símbolo es o no un metasímbolo: por ejemplo, podríamos escribir la regla anterior de alguna de las siguientes formas:

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

Un metalenguaje para la descripción sintáctica de UBL

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

1. Utilizaremos los simbolos =, (,), [,], {, } , | y % como metasímbolos; como también forman parte del lenguaje UBL, cuando sea necesario utilizarlos en este último 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 genéricas) escribiéndolas en minúsculas (aunque es posible que la primera letra sea mayúscula), y los terminales que sean identificadores (lo que llamaremos más adelante identificadores reservados y predefinidos), escribiéndolos en negrita o mayúsculas.

3. Las reglas sintacticas tendrán siempre la forma

   No_terminal = *Fórmula*

   El metasímbolo = podrá leerse "se escribe" o "se desarrolla".

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

   X = *Y Z*

   Se han visto ya varios ejemplos de esta operación.

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

   Declaración_de_objeto = *Declaración_de_variable* | *Declaración_de_constante*

6. Encerraremos una construcción entre los metasímbolos [y] cuando, en el desarrollo de determinada fórmula, pueda prescindirse de esa construcción. Por ejemplo, la sintaxis de la instrucción **si** podria ser:

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

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

7. Encerraremos una construcción entre los metasímbolos { y } cuando, en el desarrollo de determinada fórmula, pueda escribirse un número arbitrario (incluido cero) de veces esa construcción. Por ejemplo,

   {instrucción}

   representará una secuencia de cero o más instrucciones; y

   instrucción {instrucción}

   representará una o más instrucciones, con lo cual podría escribirse

   instruccion(es) = instrucción {instrucción}

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

8. Utilizaremos el metasímbolo % para encerrar meta-comentarios, descripciones en castellano que substituirán a formalizaciones mas estrictas en casos en los que sea imprescindible (un ejemplo de utilización de "%" se encuentra a continuación).

9. Por último, utilizaremos los metasímbolos (y) como meta-paréntesis para evitar ambiguedades en la interpretacion de las fórmulas.

Un ejemplo: Alfabeto del lenguaje UBL.

Como ejemplo real de aplicación del formalismo definido, damos la descripción del alfabeto básico del lenguaje UBL; contiene los caracteres que toda versión del lenguaje debe proporcionar (aunque cada versión pueda definir caracteres adicionales). Los caracteres están agrupados por categorías.

```
carácter =
  letra | dígito | carácter_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

dígito = 0|1|2|3|4|5|6|7|8|9

carácter_especial =
  "(" | ")" | "[" | "]" | "{" | "}" | + | - | * | ^ | v | ~
  | / | "|" | " | ' | ↑ | _ | < | > | : | ; | . | , | "="

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

Figura 10. Alfabeto del lenguaje UBL

Símbolos

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

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

tira = "{caracter}"

carácter_constante = 'caracter'

número_sin_signo = entero_sin_signo | real_sin_signo

entero_sin_signo = dígito{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

símbolo_especial = cáracter_especial | ← | ≤ | ≠ | ≥ | ⇒ | .. | □
```

Figura 11. ***Vocabulario del lenguaje UBL:*** Notese la introducción de no terminales auxiliares en la descripción 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 números y/o letras — no se permiten blancos intermedios, ni caracteres especiales. Puede escribirse el carácter de subrayado entre dos caracteres (pero no dos subrayados seguidos, ni uno al final), para aumentar su legibilidad. Si dos identificadores difieren sólo en que alguna letra es mayúscula en uno y minúscula en otro, se consideran iguales.

Asi,

A X Universidad_de_Barcelona Ceñudo X25 X_1_1_1

son identificadores válidos;

resultado Resultado RESULTADO ReSuLtAdO

son cuatro modos de escribir el mismo identificador; y

Universidad de Barcelona
A__Y__B
1X2
*A_*
Mas_o_-

no son [válidos 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 demás, no pueden utilizarse libremente asociándolos con abstracciones, sino que cumplen determinado papel en la sintaxis y semántica de un programa. Se les llama ***identificadores reservados*** o ***palabras clave***; en este manual se representarán siempre en negrita.

Asimismo, determinadas abstracciones (como los tipos *entero* y *caracter* o las acciones *lee* y *escribe*) están definidas automáticamente. Llamaremos ***identificadores predefinidos*** a los nombres que las representan. Al final del capítulo 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 ningún 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 comprenderá mejor más adelante, al hablar de declaraciones y reglas de reconocimiento de nombres).

Caracteres y tiras de caracteres: Las tiras (de caracteres) representan información alfanumérica constante de cualquier longitud (número de caracteres) y se "encierran" entre comillas. Pueden contener caracteres del alfabeto básico (incluido el espacio en blanco) y otros que posea el ordenador que se utilice. Para evitar ambigüedades, si alguno de los caracteres de la tira es una comilla ("), ésta se escribirá dos veces: así, 'Dijo: "Cállate"' se escribirá

```
"Dijo: ""Cállate""".
```

Una tira también puede no contener ningún carácter (en cuyo caso se representa ""); la llamaremos ***tira vacía.*** Una tira debe estar siempre contenida en una línea; es decir, no son posibles construcciones del tipo

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

Un carácter constante es un carácter encerrado entre apóstrofes y representa información alfanumérica de longitud 1. Se aplican las mismas observaciones que a las tiras; en particular, el carácter "'" se escribirá ''''.

Números: El lenguaje UBL proporciona dos tipos de datos (además de los definibles por el programador) para tratar información numérica: el tipo *entero* (que ya se ha estudiado), y el tipo *real* (que se verá más adelante).

Los enteros sin signo representan números enteros no negativos; cada versión de UBL puede establecer un máximo y un mínimo en el conjunto de enteros aceptables. Son ejemplos de "entero_sin_signo"

1
23
77
12345678

Los reales sin signo representan números reales, escritos en notación exponencial (la E se lee "por diez elevado a"). Se distinguen de los enteros en que se escriben con punto decimal y/o exponente. Nótese que, de escribirse el punto decimal, éste debe ir seguido de al menos un dígito. Cada versión puede imponer límites en cuanto al número de dígitos significativos y la magnitud del exponente.

Son *reales_sin_signo*

1.2
3.1415926535
1.11111E+23
23E-23

pero no

1.
.23
E10

Símbolos especiales Los símbolos especiales incluyen separadores ("..","(",")",",",";",":","|"), utilizados como "signos de puntuación" del lenguaje, y operadores (como "+" o "-").

Comentarios

En ocasiones, puede ser necesario tener información relativa a un programa (explicación del algoritmo utilizado, autor y fecha del programa, que representa cada variable). Esto puede hacerse mediante algún mecanismo de documentación externa (creando un fichero que contenga esa información) 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 información que proporciona. Por ejemplo, una declaración 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 línea,[9] 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 único sentido es el de documentar un programa.

Dos errores frecuentes al utilizar comentarios son la ***subdocumentación*** y la ***sobredocumentación***: un programa (especialmente si es largo) pobremente documentado será difícilmente comprensible; y demasiados comentarios en un programa oscurecerán la comprensión de éste: debe procurarse, mediante la elección de identificadores mnemotécnicos y la distribución adecuada del texto, eliminar la documentación 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:

---

[9] Debe entenderse que desde la aparición de "--" hasta fin de línea se toma el texto como comentario; no es posible utilizar el simbolo "--", poner luego un comentario, volver a poner "--", y seguir con el programa, todo en la misma línea.

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

Además, el final de una línea se intrepreta como un espacio en blanco adicional, lo cual impide "partir" un símbolo entre el final de una línea y el principio de otra.

Asi, el Algoritmo 3 en la página 16 podría 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 número 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 difícilmente comprensible (si lo es en absoluto), aparte de ser estéticamente horrible. Para evitar en lo posible la proliferación de tales programas, sugeriremos diversos estilos de escritura para cada una de las construcciones del lenguaje; aunque estas convenciones se explicarán 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 intérprete automático, sino únicamente destinadas a facilitar la legibilidad, mantenimiento e intercambiabilidad de los algoritmos.

- Llamaremos ***indentación***[3] de una línea en un programa al número (eventualmente nulo) de espacios en blanco que la preceden (este concepto es conocido en castellano bajo el nombre de ***sangrado***). Igualmente, diremos que una línea está "indentada tres espacios", o bien que su ***nivel de indentación*** es tres. El uso cuidadoso de la indentación será la base de nuestras convenciones estilísticas.

  Existe una subordinación lógica entre las instrucciones (y declaraciones; como se verá) de un programa: si una instrucción *I* debe repetirse hasta la verificación de una condición *C*,

  ```
  repite
    I
  hastaque C;
  ```

podemos decir que *I* es una instrucción subordinada (o sub-instrucción) de la instrucción **repite** completa. Expresaremos esto mediante la indentación de la instrucción *I* un número fijo de espacios (en este manual utilizamos dos). Igualmente, si la repetición debiera ejecutarse solo en el caso del cumplimiento de una condición *D*, escribiríamos

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

- En el caso de que una instrucción no compuesta o una declaración ocupe más de una linea, las lineas adicionales se distinguirán mediante una indentación adicional. Ejemplos:

```
r ← 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 continuación la sintaxis y los modelos de indentación sugeridos para las construcciones explicadas; las variantes propuestas abarcan todos los casos posibles.

Sintaxis

instrucción_si =
si *condición* **entonces**
instrucción
{*instrucción*}
[**sino**
instrucción
{*instrucción*}]
fin [**si**];

Estilo

Caso general:

si *condición* **entonces**
instruccion(es)
sino
instruccion(es)
fin si;

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

si *condición* **entonces**
instruccion(es)
fin si;

Si la(s) instruccion(es) caben en una sola línea, puede utilizarse

si *condición* **entonces** *instruccion(es)*
sino *instruccion(es)*
fin si;

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

Si la instrucción **si** entera cabe en una sola línea, puede utilizarse

si *condición* **entonces** *instrucción* **sino** *instrucción* **fin**;

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

***Figura 12. Sintaxis y estilos de escritura para la forma simple de la instrucción* si**

Los estilos mas compactos se utilizarán sólo si se desea; siempre es posible utilizar el descrito en el caso general. Lo mismo se aplica a las demás convenciones estilísticas, que, por otra parte, se describen más informalmente que la sintaxis.

```
instrucción_de_asignación = variable ← expresión;
```

Figura 13. Sintaxis de la instrucción de asignación

Sintaxis

instrucción_repite =
  repite
    instrucción
    {*instrucción*}
  hastaque *condición;*

Estilo

Caso general:

```
repite
  instruccion(es)
hastaque condición;
```

Si la repetición entera cabe en una sola linea:

```
repite instrucción hastaque condición;
```

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

```
programa =
  programa identificador es
    declaración
    {declaración}
  haz
    instrucción
    {instrucción}
  fin programa;
```

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

Sintaxis

declaración_de_objeto =
declaración_de_constante
| declaración_de_variable

declaración_de_variable = **[var]** *identificador {,identificador}: tipo;*
declaración_de_constante = **const** *identificador "=" expresión_constante;*

Estilo

var *identificador(es): tipo; {identificador(es): tipo}*
const *identificador = expresión_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 escribirán siempre cada una en una linea.

Ejercicios

1. Escribir las expresiones resultantes de

   q = [1] [X] [2]

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

3. Los ceros a la izquierda en los números enteros no son significativos. Escribir un fragmento modificado de la sintaxis del lenguaje UBL que no permita los ceros a la izquierda. **[Indicación:** dividir la categoría sintáctica *dígito* en dos, como *digito_no_nulo* y *cero*].

Apéndice: tablas de identificadores reservados y predefinidos en UBL catalán y castellano.

acaba	**es**	**mod**	
accion	**existe**	**modulo**	**sal**
			si
ciclico	**fila**	**nada**	**sino**
con	**fin**	**nombre**	**sucesion**
condicion	**funcion**	**nulo**	
conjunto			**tabla**
const	**hastaque**	**otros**	**talque**
	haz		**tipo**
de		**para**	**tupla**
div	**implementa**	**produce**	**usa**
	itera	**programa**	
en			**vale**
entonces	**mientras**	**repite**	**var**

Figura 17. Tabla de identificadores reservados en UBL Castellano

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 18. Tabla de identificadores predefinidos en UBL Castellano.

acaba	**es**	**mentre**	**si**
accio	**existeix**	**mod**	**sino**
altres		**modul**	**successio**
amb	**fes**		**surt**
	fi	**nom**	
ciclic	**fila**	**nul**	**talque**
condicio	**finsque**		**taula**
conjunt	**funcio**	**per**	**tipus**
const		**produeix**	**tupla**
	implementa	**programa**	
de	**itera**		**usa**
div		**repeteix**	
	llavors	**res**	**val**
en			**var**

Figura 19. Tabla de identificadores reservados en UBL Catalán

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

Figura 20. Tabla de identificadores predefinidos en UBL Catalán.

Diseño descendente. Acciones y Condiciones

Los conceptos introducidos en capítulos anteriores permiten el diseño de programas arbitrariamente complejos; sin embargo, no hemos presentado ningún método para su elaboración, sino que hemos sugerido. mostrándola en la practica con varios ejemplos, la conveniencia de *análisis teóricos* y *expresiónes en el lenguaje* de esos análisis. Describiremos detalladamente una aproximación a la construcción sistemática de algoritmos que se conoce como ***diseño descendente*** (en inglés, *top-down design*) o ***método de refinamiento progresivo***, así como algunas partes del lenguaje UBL adecuadas a ese método.

Un ejemplo de diseño descendente

Disponemos de un texto formado por frases no vacías acabadas en un punto; el texto mismo se termina con un asterisco que sigue inmediatamente al último punto. Deseamos escribir un programa que cuente el número medio de As que aparecen en las frases del texto. Una primera aproximación podría ser:

```
-- Número Medio de As. Refinamiento 1 --
programa numero_medio_de_As es
haz
  preséntate;
  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 estaría terminado. Este no es el caso, de modo que será necesaria una mayor elaboración de cada instrucción hasta hacerla comprensible por el intérprete; el programa mostrado puede tomarse, de todos modos, como una primera versión o esquema del algoritmo completo. La construcción del programa pasará ahora por un ***refinamiento*** o detalle de sus instrucciones y condiciones, que

podrán considerarse, hasta cierto punto, como (sub)algoritmos a desarrollar independientemente.

Cada desarrollo nos conducirá a una nueva versión del programa, que así habremos expresado según diferentes ***niveles de abstraccion***, al variar el detalle y precisión de sus instrucciones: la primera versión será la más abstracta — probablemente, también la más comprensible para el lector y la que menos suposiciones no relativas al problema hace — —, y las sucesivas ganarán en detalle y perderán en generalidad.

Encontramos aquí 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 ***división*** del problema en subproblemas posiblemente más sencillos e independientes.

Para terminar el programa, podemos adoptar varias estrategias; la más sencilla será aprovechar el esquema del Algoritmo 3 en la página 16 para refinar *cuenta_las_As_de_una_frase*. Esto nos obligará a declarar variables *c* y *n* como en ese programa; además, necesitaremos variables para acumular el número total de As y el número de frases; las llamaremos respectivamente *t* y *f*. El desarrollo del resto del programa es trivial, comparando también la estructura general con el esquema del Algoritmo 3:

```
-- Número Medio de As. Refinamiento 2a --
programa número_medio_de_As es
  var c: caracter; n, t, f: entero;
haz
  -- Inicialización --
  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' ∨ 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 número medio de As por frase es: ",t div f;
fin programa;
```

Otra solución al mismo problema. Forma general de la instrucción si. Instrucción nada

Una consideración más atenta del problema permite ver que sólo se precisa contar el número de As y el número de puntos del texto (ya que lo suponemos bien escrito), y que para ello basta con una sola iteración; cada una de ellas deberá actuar de modo distinto según el carácter examinado sea un punto, una A u otro cáracter: se trata de una toma de decisión con más de dos alternativas, que podría escribirse utilizando combinaciones de instrucciones **si**:

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

Este tipo de construcción, como se verá, es bastante frecuente, y resulta poco clara escrita de este modo. Introduciremos pues una nueva forma, más general, de instrucción **si**, mediante la cual lo anterior podrá expresarse

```
si
  □ (c = 'A') ∨ (c = 'a')  ⇒ ...
  □ (c = '.')  ⇒ ...
  □ otros  ⇒ ...
fin si;
```

La forma general de la instrucción **si** es

```
instrucción_si =
  si
    □ condición ⇒
      instrucción
      {instrucción}
    {□ condición ⇒
      instrucción
      {instrucción}}
    /□ otros ⇒
      instrucción
      {instrucción}/
  fin /si/;
```

Figura 21. ***Sintaxis y estilo de la forma general de la instrucción*** **si:** si la (o las) instruccion(es) asociadas a una alternativa caben a continuación de la flecha ⇒, pueden escribirse juntas en esa línea.

Efecto de la ejecución de la instrucción si en su forma general: se evalúan secuencialmente las condiciones escritas entre □ y ⇒, en el orden en que estan escritas, hasta encontrar una que sea cierta, en cuyo caso se ejecutan las instrucciones asociadas. Si ninguna condición es cierta, se ejecutan, si existen, las instrucciones asociadas a □ **otros**; si se omite □ **otros**, se produce un error si no se verifica ninguna de las condiciones. Una vez determinada la alternativa a ejecutar, no se evalúa ninguna otra condición.

Dado que se evalúan todas las condiciones hasta encontrar una que sea cierta, será conveniente de cara a la eficiencia del programa escribir las condiciones en orden decreciente de probabilidad, con objeto de que la máxima evaluación se produzca en el mínimo número de casos.

Definimos también una instrucción **nada**, de efecto nulo, que sera útil entre otras cosas en el tratamiento de determinadas alternativas que no requieren ejecución alguna:

```
instrucción_nada = nada;
```

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

Utilizando estas definiciones, podemos considerar la forma simplificada de la instrucción **si** como un caso particular de su forma general: en efecto, si el número de condiciones es dos

```
si
  □ condición1 ⇒ instrucciones1
  □ condición2 ⇒ instrucciones2
fin si;
```

y además se verifica

$$condición_1 = \sim condición_2$$

de modo que también hubiese podido escribirse

```
si
  □ condición1 ⇒ instrucciones1
  □ otros ⇒ instrucciones2
fin si;
```

o

si
  □ *condición*$_2$ ⇒ *instrucciones*$_2$
  □ **otros** ⇒ *instrucciones*$_1$
fin si;

permitiremos la abreviación

si *condición*$_1$ **entonces** *instrucciones*$_1$
sino *instrucciones*$_2$
fin si;

y también la

si *condición*$_2$ **entonces** *instrucciones*$_2$
sino *instrucciones*$_1$
fin si;

permitiendo la omisión de **sino** cuando su instrucción subordinada sea **nada**.

Volviendo al programa que nos ocupa, podemos ver que las tres instrucciones contenidas dentro de la iteración del Refinamiento 1 se "funden", al elaborarlas, en una sola instrucción **si**, 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 método de diseño descendente: es posible que un refinamiento obligue a replantear el esquema del cual proviene. El programa completo podrá escribirse:

```
-- Número Medio de As. Refinamiento 2b --
programa número_medio_de_As es
  var c: caracter; t, f: entero;
haz
  -- Inicialización --
  escribe_linea "Escribe un texto terminado por un asterisco";
  t ← 0; f ← 0;
  -- Proceso del texto --
  repite
    lee c;
    -- selección de caracteres --
    si
      □ (c = 'A') v (c = 'a')  ⇒ t ← t + 1;
      □ c = '.'  ⇒ f ← f + 1;
      □ otros ⇒ nada;
    fin si;
  hastaque c = '*';
  -- escribe resultados --
  escribe_linea "El número 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 versión es más corta y eficiente que la primera (2a), al haberse escrito pensando la solución sin recurrir a resultados establecidos. También se notará que este programa no requiere que el asterisco final siga inmediatamente al último punto, así como el orden en que se han escrito las condiciones, basado en la suposición razonable de que cada frase contendrá varias letras A. Se ha utilizado "□ **otros ⇒ nada**;" para evitar errores de ejecución al procesar caracteres distintos del punto y la A.

Acciones y Condiciones. Instrucciones vale y acaba

El método expuesto tiene el inconveniente de que la versión final del algoritmo no refleja el proceso de diseño a que ha sido sometido, que en ocasiones interesa más en el problema que el propio programa. Para evitar esta pérdida de información, el lenguaje UBL proporciona dos mecanismos de declaración de entidades, que llamaremos **acciones** y **condiciones**, para aumentar el universo léxico disponible mediante la introducción 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 información contenida en el 1. Podemos hacer esto instruyendo al intérprete, mediante declaraciones apropiadas, sobre cuál es el significado de instrucciones como *preséntate*:

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

Esta declaración introduce una nueva abstracción que permite utilizar *preséntate;* como una nueva instrucción (en este caso, definida por nosotros). Igualmente, podemos indicar el sentido de *se_termine_el_texto* mediante la declaración

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

Mientras que la declaración de *preséntate* es una instrucción abstracta, *se_termine_el_texto* es una abstracción de evaluación: puede contener instrucciones como en la **accion** anterior, pero debe incluir (y ejecutar) al menos una instrucción **vale** (que expresa cuál es el valor asociado a la abstracción que representa y termina la ejecución de ésta).

instrucción_vale = **vale** *condición;*

Figura 23. Sintaxis de la instrucción* vale *en condiciones

Utilizando este método, obtenemos el siguiente programa:

```
-- Número Medio de As. Refinamiento 2c --
programa número_medio_de_As es

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

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

  accion cuenta_las_As_de_una_frase haz
    n ← 0;
    repite
      lee c;
      si c = 'A' ∨ 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 número medio de As por frase es: ",t div f;
  fin escribe_resultados;

haz
  preséntate;
  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 más largo, pero puede argumentarse que es también más comprensible. La efectividad de la notación quedará clara, de todos modos, en programas más complejos.

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

declaración_de_acción =
  accion *identificador* **haz**
    instrucción
    {*instrucción*}
  fin [*identificador*];

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

accion *identificador* **haz** *instrucción* {*instrucción*} **fin**;

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

Una declaración de este estilo establece una asociación entre el identificador y las instrucciones, de modo que la aparición del identificador en el programa indica la ejecución de las instrucciones asociadas. Llamaremos a tal instrucción ***invocación*** a una acción:

instrucción_de_invocación = *identificador*;

Figura 25. Sintaxis de la instrucción de invocación (Simplificada)

Las condiciones se declaran de modo similar:

declaración_de_condición =
  condicion *identificador* **haz**
    instrucción
    {*instrucción*}
  fin [*identificador*];

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

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

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

En este caso se asocia el identificador con la ejecución de las instrucciones asociadas, que debe terminarse con la de una instruccion **vale**: ésta proporciona el

valor de la condición, que puede utilizarse también en expresiones más complejas. En cuanto a las acciones, puede utilizarse la instrucción **acaba** para terminar incondicionalmente la ejecución del subprograma:

instrucción_acaba = **acaba**;

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

Resumiendo: las **accione**s y **condicione**s declaran abstracciones (de instrucción y evaluación, respectivamente). La aparición en el programa del identificador que las representa tiene como efecto la ***activación*** del respectivo ***subprograma*** (que también llamaremos ***bloque***) y la ejecución de las instrucciones asociadas. Durante la ejecución de esas instrucciones, diremos que el bloque está ***activo***; dejara de estarlo al ***terminar*** esa ejecución, cosa que puede suceder naturalmente (en el caso de una **accion**) o como efecto de la ejecución 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 ***colisión 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 página 16 utilizando la técnica de diseño descendente en los dos modos mostrados en este capitulo. Hacer lo mismo con los demás programas realizados hasta el momento.

2. El Máximo Comun Divisor (*mcd*) de dos números positivos es el mayor número que los divide exactamente a ambos. Diseñar un algoritmo que calcule el mcd de dos números 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. Dados un dígrafo y un texto, calcular la media de apariciones de ese dígrafo por frase.

4. Dada una serie ascendente de números 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 información alfanumérica a nivel complejo: en aplicaciones de tratamiento de textos, puede necesitarse trabajar con palabras o frases, tal como se utilizan los enteros en aplicaciones numéricas o los caracteres en proceso elemental de textos. Para ello se introduce el tipo *tira*, del cual son literales, como ya se dijo, los símbolos definidos como "*tira*" en el vocabulario y escritos entre comillas.

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

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

son tiras de caracteres (recordamos que se conoce como ***tira vacía*** 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 máxima*** de los valores que puede tomar la variable:

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

Hablaremos también de la ***longitud actual*** de una variable de tipo tira, indicando la longitud de su valor, para distinguirla de la longitud máxima o declarada. Así, podemos formular la siguiente

Definición: Llamaremos ***error de truncación*** al que ocurre al intentar asignar a un objeto de tipo tira un valor cuya longitud actual es mayor que su longitud máxima.

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 = "Introducción a la Programación en UBL";
```

Operaciones sobre tiras

Además de la asignación, distinguiremos las siguientes operaciones básicas sobre tiras (de las cuales pueden deducirse las demás):

- La ***concatenación*** 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 ***selección*** de un caracter, que permite obtener el valor de uno de los caracteres individuales que componen la tira: se expresa la posición o ***índice*** 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 índice 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 índices:

  ```
  t ← "ABCDEFGH";
  -- t[1..3] = "ABC";
  -- t[3..2] = ""
  -- t[4..4] = "D" (distinto de T[4] = 'D', que es un carácter)
  ```

  Los índices están 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 índice (o ***límite superior***) y el primero (o ***límite inferior***) menos 1. Si el límite superior es menor que el límite inferior, convendremos en que el resultado es la tira vacia.

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

  t ← "123"; -- long(t) = 3

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

  tira(c)

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

Comparacion de tiras

La comparación de objetos de tipo tira se efectua utilizando la ***ordenacion lexicográfica***, alfabética 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 relación 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 (téngase también 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 ordenación utilizada habitualmente en los diccionarios.

2. Al comparar dos tiras de diferente longitud, se considerará que la más 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 comparación, `"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 instrucción *escribe* (o *escribe_linea*), aplicada a una (expresión de tipo) tira, se ha utilizado ya en el caso de tiras constantes; sólo hay que añadir que se escriben tantos caracteres como tenga la tira en ese momento (y no tantos como su longitud máxima).
- La instrucción *lee* acepta tantos caracteres como la longitud máxima de la tira leída (o, si no los hay hasta fin de línea, los que haya), y forma con ellos una tira que asigna a la variable que se lee. Nunca se cambia de línea por efecto de una lectura de *tira*.
- Para efectuar un cambio de línea al leer datos, puede utilizarse la instrucción *lee_linea* (seguida o no de objetos a leer), cuyo efecto es ignorar (después de leer los objetos designados) el resto de la línea 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 descomposición 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 índice *i* que "recorrerá" 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 demás variables enteras
  -- identifican los subíndices 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. ***Descomposición en Nombre y Apellidos:*** Nótese 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 operación de lectura; el acceso posterior (y secuencial) a las componentes de lo leído no debe confundirse con ella.

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

El segundo programa, que puede tener más aplicación práctica, intenta hallar el número 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 línea; las líneas 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 mínimo las siguientes variables: dos tiras (que podemos llamar *p* y *q*) para representar la palabra que se busca y cada una de las demás, 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 "Qué palabra quieres buscar?";
lee p;
escribe_linea "Escribe ahora el texto, terminado por un asterisco";
lee_linea;
```

aprovechando para informar de qué deseamos. Se notara la utilización de *lee* y *lee_linea*, forzada por el hecho de que esta última 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 serán 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 puntuación (Para simplificar, supondremos que los únicos signos de puntuación son el punto, la coma, los dos puntos y el punto y coma). Así, añadiremos una variable

```
c: caracter;
```

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

```
q ← "";
repite
  q ← q + tira(c);
  lee c;
hastaque (c = ' ') ∨ (c = '.') ∨ (c = ',') ∨ (c = ':') ∨ (c = ';') ∨ (c = '*');
si (c = ' ') ∨ (c = '.') ∨ (c = ',') ∨ (c = ':') ∨ (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 puntuación, y manteniendo la hipótesis de que *c*, al terminar la acción, 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 refinará 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 será

```
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 = ' ') ∨ (c = '.') ∨ (c = ',') ∨ (c = ':') ∨ (c = ';') ∨ (c = '*');
    si (c = ' ') ∨ (c = '.') ∨ (c = ',') ∨ (c = ':') ∨ (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 página 57 para

   a. evitar la suposición de que un blanco sigue al segundo apellido
   b. evitar la repetición de instrucciones **repite**.

3. Escribir un programa que detecte la existencia de palabras palindrómicas [esto es, simétricas: como ALA ASA AMA ATA POP GAG u otras de mayor longitud].

4. Hacer lo mismo con frases palindrómicas, como *dabale arroz a la zorra el abad*. Suponer un límite 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 posición 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 numéricas: el tipo *entero* (que ya se ha estudiado) y el tipo *real*; intentan reflejar la estructura de los conjuntos Z y R de la Matemática, respectivamente. Los números reales pueden tener decimales (hasta un máximo dependiente de versión) y estar expresados en notación exponencial (lo cual permite abarcar un amplio rango de números aunque haya pocos dígitos significativos). Se ha descrito ya la sintaxis de los literales reales; una declaración de objeto real tiene la forma acostumbrada:

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

Las operaciones aplicables a los números reales son la suma (+), resta (-), multiplicación (*) y división (/). Se pueden mezclar reales y enteros en las operaciones aritméticas; el resultado siempre es real. Asimismo, pueden dividirse dos enteros mediante el operador real "/"; en tal caso, el resultado es también real.

```
2.0 + 3.0 = 5.0
3.5 - 2 = 1.5
1.0/3.0 = 0.333333 (con más o menos precisión)
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 formación de caracteres que responda a la sintaxis de un entero o de un real; en escritura, se escribe el número en notación exponencial.

Precisión de los objetos de tipo *real*

Los números reales se representan en el ordenador con un número fijo y acotado de dígitos. Como consecuencia de ello, muchos de los resultados y teoremas matemáticos dejan de ser válidos:

- Supongamos que el número de dígitos que maneja el ordenador es 6. Entre los números 1.00000 y 1.00001 no hay ninguno intermedio ***que tenga sólo seis dígitos***: los reales representables no forman un conjunto continuo.

- La igualdad $r+s=s$ no permite deducir $s=0$. Basta verlo con los números $r=1.0$ y $s=1.0E-10$:

  al sumarlos se obtiene

  $$\begin{array}{rl} & r = 1.0000000000 \\ + & s = 0.0000000001 \\ \hline & r+s = 1.0000000001 \end{array} \rightarrow \textit{truncado a 6 digitos: } 1.00000 = 1 = r,$$

- $10/3*3 \neq 10$, 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 límites que se consideren oportunos: si en la instrucción

si $x = y$ **entonces** *escribe "La solución 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

$abs(x - y) < 1E-5$

Donde *abs* es una función predefinida que calcula el valor absoluto. Además, el exponente está también limitado. Si se intenta leer un número que exceda a su máximo, o alguna operación produce un resultado que está fuera del rango permitido, se produce un error.

Ejemplo: diferencias de cálculo en la serie armónica

La serie armónica se define como

$h_n = 1 + 1/2 + 1/3 + \ldots + 1/n$

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 (teóricamente deberían ser idénticos), que son relativamente más importantes al crecer n.

```
programa serie_armónica 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 armónica

El tipo *logico*

El tipo *logico* se define por su conjunto de valores (sólo tiene dos, que llamaremos *cierto* y *falso*) y las operaciones aplicables (∧, ∨, ~, ya estudiadas; ∧∧,∨∨, que se verán a continuación). 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 condición en determinado momento de la ejecución del programa. Debe evitarse utilizar este recurso si es posible.

```
l ← x < 0 ∧ (a > b ∨ a < c);
...
si l entonces ...
```

Más adelante, al estudiar tipos de datos estructurados, se verán otras aplicaciones de este tipo.

Operadores lógicos 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 producirá un error si, en el momento de evaluarse la condición, la tira *t* no tiene una longitud mayor o igual que tres. Para prever este caso, podríamos intentar una solución del estilo de

si *long(T) ≥ 3 ∧ T[1..3] = "ABC" ...*

pero, dado que el significado del operador "∧" especifica que deben evaluarse primero los dos operandos y después hacer el "∧" lógico, nos encontramos con el mismo problema. Una solución correcta sería

si *long(T) ≥ 3* **entonces si** *T[1..3] = "ABC" ...*

aunque esto complicaría el programa, al añadir una nueva instrucción **si**.

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

A ∧∧ B *(se lee "**y entonces**")*
A ∨∨ B *(se lee "o **sinó**")*

que se evaluan como sigue:

A	*B*	*A ∧∧ B*
Falso	*No se evalúa*	*Falso*
Cierto	*Falso*	*Falso*
Cierto	*Cierto*	*Cierto*

A	*B*	*A ∨∨ B*
Cierto	*No se evalúa*	*Cierto*
Falso	*Falso*	*Falso*
Falso	*Cierto*	*Cierto*

Figura 28. Operadores lógicos condicionales

"No se evalúa" 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 expresión

lógica se conoce sin necesidad de hacerlo: *falso* ∧ *x* es falso, independientemente de x, y *cierto* ∨ *x* es siempre cierto)

Mediante estas formas, el fragmento anterior podra escribirse correctamente como

si *long(T)* ≥ *3* ∧∧ *T[1..3]* = *"ABC"* ...

limitando el cálculo de *T[1..3]="ABC"* al caso en que *long(T)≥3*.

Expresiones

Podemos definir ahora con más precisión qué entendemos por una ***expresión***: en general, se compone de varios operandos y operadores; su ***evaluación*** 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 "más fuerte" -- que "+"). Se supone también que todos los operadores son ***asociativos por la izquierda***: esto es,

a + b + c + d

se interpreta como

((a + b) + c) + d

El formalismo sintáctico definido en "Sintaxis y estilo" en la página 27 nos permitirá describir la forma general de una expresión:

```
expresión = relación { ∧ relación }
  | relación { ∨ relación }
  | relación { ∧∧ relación }
  | relación { ∨∨ relación }

relación = expresión_simple [ operador_relacional expresión_simple ]

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

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

operador_aditivo = + | -

término = factor { operador_multiplicativo factor }

operador_multiplicativo = * | / | div | mod

factor = número_sin_signo
  | tira
  | carácter_constante
  | conjunto
  | existencial
  | variable
  | selección_de_tira
  | invocación_de_función
  | invocación_de_condición
  | conversión_de_tipo
  | "(" expresión ")"
  | ~ factor

selección_de_tira = variable "[" expresión [ ..expresión ] "]"
```

Figura 29. Sintaxis de una expresión

La prioridad de los distintos operadores queda reflejada en la sintaxis, así como el papel de los paréntesis para modificar el orden de evaluación.

Los no terminales *conjunto*, *existencial* e *invocación_de_funcion*, así como el operador relacional "**en**", se estudiarán más adelante; *variable* puede ser una variable (de las ya estudiadas) o asumir formas más complejas (que también se estudiarán). *Invocación_de_condicion*, puede ser, en su forma más simple, el nombre de una condición. En cuanto a *conversión_de_tipo*, se refiere a las diversas posibilidades de conversion entre tipos que ofrece el lenguaje UBL. Se ha visto ya la conversión a tira:

tira(expresión_caracter)

Otros ejemplos se estudiarán más adelante.

Parámetros 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 ámbito lógico puramente ***local*** (queremos decir: circunscrito a un solo subprograma). En este último caso, declararlas en el programa principal causa problemas de estructura y reduce el espacio de identificadores disponible. El concepto de ***declaración local*** sirve para evitar estos problemas, al permitir asociar cada declaración con el ámbito mínimo que lógicamente le corresponde:

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

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

Accesibilidad y existencia de las entidades locales

Un objeto variable declarado localmente sólo existe con cada ejecución del subprograma que lo declara: en el ejemplo anterior, la variable *t* es "creada" cada vez que comienza la ejecución de *intercambia* y "destruida" cada vez que termina ésta. De este modo, es necesario asignar algún valor a las variables locales antes de utilizarlas: es un error creer que conservarán su valor entre una ejecución y otra, ya que su existencia es discontinua. Además, 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 convención: 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 aquél en el que se estaba buscando; y así, sucesivamente, en todas las listas de todos los subprogramas que contienen sintácticamente 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 acción A
       accede a la declaracion {2}, mientras que
       las referencias a J acceden a la declaración {1} *)
  fin a;
haz -- programa
  (* cualquier referencia a I o j en el programa accede a la declaración {1} *)
fin programa;
```

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

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

```
accion A; -- anuncia que A se declarará más abajo, aunque ya puede utilizarse

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

accion A haz -- ésta es la declaración real de A
  ...
fin A;
```

Parámetros

La potencia de las **acciones** y **condiciones** (y de los demás tipos de subprograma que se estudiarán) se amplia notablemente si consideramos la posibilidad de ***parametrizar*** sus efectos: la acción *Intercambia* del ejemplo anterior será mucho más util y general si se permite utilizarla en la forma

```
intercambia i,j;
```

o

```
intercambia k,p;
```

según cuales sean las variables que se quieran intercambiar; una condición *es_mayúscula* podrá aplicarse en más casos si admite un parámetro:

```
es_mayúscula(c)
```

Un subprograma declara sus parámetros inmediatamente después del identificador que lo designa, y entre paréntesis:

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

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

Distinguiremos varios tipos de parámetros:

```
lista_de_parámetros = parámetros {; parámetros}

parámetros = parámetros_por_copia
  | parámetros_por_nombre
  | parámetros_por_subprograma

parámetros_por_nombre = parámetros_por_variable | parámetros_por_constante

lista_de_identificadores = identificador {,identificador}

parámetros_por_copia = lista_de_identificadores: identificador;

parámetros_por_variable = var lista_de_identificadores: identificador;

parámetros_por_constante = const lista_de_identificadores: identificador;
```

Figura 30. ***Sintaxis de la lista de parámetros:*** Los *parámetros_por_subprograma* se estudiarán mas adelante.

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

- Un ***parámetro 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 invocación como

  ```
  raya 30;
  ```

  ejecuta la acción *raya* después de asignar el valor del argumento (*30* en este caso) al parámetro *n*.

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

- Un ***parámetro por variable*** funciona como un ***sinónimo*** para la variable proporcionada como argumento:

  La acción *intercambia* se generaliza mediante parámetros 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 sinónimos locales de los argumentos proporcionados: la ejecución de la última instrucción será equivalente a

```
t ← i; i ← j; j ← t;
```

que tiene el efecto buscado.

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

- Un ***parámetro por constante*** es un ***sinónimo*** que funciona como un objeto ***constante*** con el valor de la expresión suministrada como argumento. Consecuentemente, no es posible alterar dentro de un subprograma (mediante una asignación u otros medios) el valor de un parámetro constante, y es un error establecerlo como sinónimo de un argumento que va a ser variado durante la ejecución del subprograma.

```
var x: entero;

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

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

Suelen utilizarse parámetros por constante cuando su tipo es estructurado (p. ej., tiras, o **tablas** o **tuplas**, que se estudiarán) 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 utilización de parámetros pueden encontrarse en los siguientes capítulos.

Efecto de Alias

Diremos que se produce un ***efecto de alias*** cuando, debido a las asociaciones introducidas por los parámetros 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 suposición 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 además ilustra un principio más general: otro modo de escribir la acción *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 demostración de la validez del método. Se observará que el cálculo 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 (obsérvese que esto no sucede con la otra versión de *intercambia*).

Sintaxis completa de las declaraciones de subprograma y sus invocaciones

```
declaración_de_subprograma = cabecera bloque | cabecera;

bloque =
  [es
    {declaración}]
  haz
    instrucción
    {instrucción}
  fin [identificador];

cabecera =
    accion identificador ["("lista_de_parámetros")"]
  | condicion identificador ["("lista_de_parámetros")"]
  | funcion identificador ["("lista_de_parámetros")"]: identificador
  | sucesion identificador ["("lista_de_parámetros")"]: identificador
```

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

```
argumentos = argumento {,argumento}

argumento = expresión
```

Figura 32. Sintaxis de la lista de argumentos

```
instrucción_de_invocacion = identificador [argumentos];
```

***Figura 33. Sintaxis de la instrucción de invocación a una* accion**

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

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

Funciones

Introduciremos un nuevo tipo de subprograma que nos permitirá escribir abstracciones de evaluación.

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 transformación de datos (p. ej., la raíz cuadrada de un número, 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 lógicos. La mayoria de funciones tienen parámetros por copia o constante que corresponden a las variables independientes de las funciones matemáticas y a partir de los cuales se evalúa el resultado. Estos parámetros se declaran con la función:

lo que matemáticamente se expresa

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

se escribirá en UBL como

funcion *f (x: S): T*

por ejemplo,

funcion *impar(n: entero): logico*

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

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

es una ***declaración de función***, que introduce una abstracción 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 ***invocación de función***:

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

media(z1,z2) o *media(x,3.0)* son invocaciones de función. *A* y *b* no son más que ***parámetros*** o "variables falsas" que toman los valores indicados entre paréntesis en cada invocación: 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.

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

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

Al igual que en las **condiciones**, la ejecución de una función debe terminar con la de una instrucción **vale**, que en este caso deberá designar una expresión del tipo declarado con la función e indicará el valor que toma la expresión que denota su invocación.

instrucción_vale = **vale** *expresión*;

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

Pueden declararse tantos parámetros como sea necesario; los tipos de éstos 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 *entero*, calcula el valor absoluto de una expresión.
- **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 número *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 declaración de tipo para *tiras*

La sintaxis de una declaración de función, así como la de la lista de parámetros, especifica que los tipos que aparecen deben escribirse en forma de identificadores. Para poder escribir funciones que operen sobre tiras (y sobre tipos más elaborados que se estudiarán más adelante) es necesario utilizar lo que llamaremos una ***declaración de tipo***:

```
tipo identificador es tira(expresión_constante);
```

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

Una declaración de este estilo amplía 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 mnemotécnico.

Funcion *índice*

Una operación 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 índice *i* (ya que *j* se obtiene como *i+long(s)-1*). Supuesta la declaración del tipo *palabra* escrita anteriormente, buscamos una

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

que evalúe *i*; para prever el caso en que *s* no está contenida en *t*, y por razones de simetría, convendremos en que

indice(s,t) = 0 ⇔ S no forma parte de T

Una primera aproximación a la escritura de la función podría 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 explicación del significado de la función. Cada iteración compara la tira *s* con una subtira de *t*, lo cual suele ser una operación costosa para el

intérprete. Otra versión, basada en buscar primero un carácter de *t* que coincida con *s[1]* y mirar luego si el resto también coincide, podría ser más económica, y se plantea como ejercicio.

Cálculo de la raíz cuadrada de un número real mediante el método de bipartición

Se trata de escribir una

funcion *raiz(r: real): real*

que calcule la raíz cuadrada de un número 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 método iterativo llamado de ***bipartición***: supuestos halladas dos aproximaciones al valor de *x*, x_0 y x_1, tales que x_0 "no llega" a ser la raíz 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 aritmética de x_0 y x_1, como una nueva aproximación. 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 ***diferirán tan solo en la mitad*** de lo que diferían antes. Aplicando el proceso tantas veces como sea necesario se obtendrá, o bien la solución exacta en algún momento del proceso, o bien una sucesión de intervalos encajados convergentes hacia un punto, que, de nuevo, sera la solución exacta. Teniendo en cuenta la finitud de la precisión de los números reales, es de esperar que la diferencia entre x_0 y x_1 llegue a ser menor el incremento mínimo que permitiría distinguirlos en el ordenador; lo que permitirá detener el proceso, como máximo, cuando $x_0 = x_1$ o $x_0 = x_2$.

Como ejemplo calcularemos la raíz 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 raíz cuadrada buscada, por lo que reemplaza a x_1, obteniendo $x_0 = 1$ y

$x_1 = 1.5$ como nuevo par de valores. Mostramos aquí los primeros pasos del cálculo:

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 cálculo de la raíz de 2 por bipartición

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

Para determinar el par de valores inicial, bastará observar que la raíz de r es menor que r si r es mayor que 1, y viceversa: de modo que 1 y r (o r y 1) serán siempre una aproximación inicial correcta (que sea buena no importa mucho, ya que el proceso de bipartición se encarga de mejorarla).

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

Algoritmo 13. Raíz cuadrada por Bipartición

Ejercicios

1. Reescribir la función *índice* tal como se indica en el texto.

2. Escribir una

   funcion *seno(x: real): real*

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

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

Sucesiones

El tratamiento secuencial de datos se presenta tan frecuentemente en programación, que hemos creido conveniente introducir un tipo de subprograma para manejarlo. Paralelamente a las **funciones**, que producen un solo valor, las **sucesiones** producen una serie o colección secuencial de ellos. Estas colecciones pueden ser tratadas de dos modos: aplicando algún tratamiento a cada uno de sus valores o buscando cual de sus elementos cumple determinada condición; además, es posible restringir el tratamiento sólo a subcolecciones de la sucesión.

Un ejemplo

Sea *t* una tira de caracteres, que deseamos imprimir "en vertical", esto es, un carácter por línea. En una primera aproximación, podemos formular esto así

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

utilizando la instrucción **para**; debe leerse

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

Caracteres_de_T es una ***sucesión*** de valores, que debe ser refinada según el método usual de diseño descendente:

```
sucesion 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;
  caracteres_de_T;
```

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

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

Diremos que una sucesión está ***activa*** desde que empieza su ejecución hasta que ***termina*** (lo cual puede suceder al llegar al final de la **sucesion**, mediante la ejecución de una instrucción **acaba** o implícitamente al evaluar un existencial). Al ejecutar una instrucción **produce**, pasa a ejecutarse la parte de programa que activó la **sucesion**, pero ésta permanece activa: diremos que está ***suspendida***, y que su ejecución ***continúa*** al seguir ejecutandose sus instrucciones. Mientras una **sucesion** está activa, las variables locales (y parámetros) conservan su valor, y sólo lo pierden al terminarse (pero no al suspenderse).

Las sucesiones pueden tener parámetros y declaraciones locales (como todos los tipos de subprograma).

Las sucesiones predefinidas *asc* y *desc*

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

Mediante el uso de la sucesión predefinida *asc*, el ejemplo anterior podría escribirse

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

sin programar ninguna **sucesion** y ganando en claridad.

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

La instrucción para

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

instrucción_para =
  para *variable* **en** *iteración* [**talque** *condición*] **haz**
    instrucción
    {*instrucción*}
  fin [**para**];

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

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

para *variable* **en** *iteración* [**talque** *condición*] **haz** *instruccion(es)*; **fin**;

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

Su significado ya se ha explicado en párrafos anteriores. La parte **talque**, opcional, sirve para limitar la ejecución de las instrucciones subordinadas a la verificación de determinada condición (que suele ser alguna propiedad de los elementos de la sucesión): 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 instrucción está tomada del cuantificador universal utilizado en Lógica.

Existenciales

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

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

Existe es una forma de condición general que se aplica sobre **sucesiones** 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 sucesión 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 instrucción **si** sabiendo de antemano que la condición es cierta, y con el único objetivo de lograr que la evaluación del existencial asigne a *i* el valor correcto.

Presentamos también la siguiente condición que evalúa (de un modo muy poco eficiente) si un entero es primo o no:

```
condicion primo(p: entero) es
  var q: entero;
haz
  vale ~ 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 están tomados también a imagen de los cuantificadores logicos; su sintaxis es

existencial = **existe** *variable* **en** *iteración* [**talque** *condición*]

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 utilización del predicado **existe** puede implicar gran cantidad de cálculos para evaluar un solo dato, por lo que es recomendable medir cuidadosamente su efecto antes de utilizarlo.

Otro ejemplo. Sintaxis de la instrucción produce

El criterio de variación ofrecido por la sucesión *asc* es bastante limitado: puede pensarse un uno más 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 último $i+nk$ que no supera a j. Una manera de programar tal sucesión es

```
sucesion 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 instrucción **produce** es

instrucción_produce = **produce** *expresión;*

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

Un programa complejo: el justificador de textos

Como muestra del poder de abstracción que proporcionan las sucesiones, desarrollaremos un programa bastante complejo: se trata de "editar" un texto, encolumnándolo 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 línea. Se tomará como dato el texto mismo, escrito en cualquier forma, y se producirá como resultado el texto bien impreso.

Utilizaremos la condición predefinida *fdf* (abreviación 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 en forma de sucesión; las agrupaciones de caracteres delimitadas por blancos formas palabras, que también se presentan secuencialmente; por ultimo, varias palabras se combinan con espacios en blanco para construir la sucesión final de líneas, que es lo que debemos imprimir.

Formulación del algoritmo: abandonando por una vez el esquema de diseño descendente, y ciñéndonos al análisis anterior, escribiremos una sucesión

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

que producirá los caracteres del texto; a continuación, y tras declarar

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

para poder declarar palabras (que se suponen de longitud máxima 20), escribiremos la sucesión

```
sucesion 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* actúa 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 notará que la señal para terminar una palabra es la aparición de un blanco, por lo que será conveniente añadir la instrucción

```
produce ' ';
```

al final de la sucesión *caracteres* (lo cual significa suponer un blanco más 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_línea = 80;
tipo línea es tira(max_línea);
```

y escribiremos la sucesión de líneas:

```
sucesion líneas: línea es
  var p: palabra; l: línea;
haz l ← "";
  para p en palabras haz
    si
      □ long(l) + long(p) + 1 > max_línea ⇒
        produce margina(l);
        l ← p;
      □ l = "" ⇒ l ← p;
      □ otros ⇒ l ← l + " " + p;
    fin si;
  fin para;
  produce l;
fin líneas;
```

Margina es una función que aplica líneas en líneas y se limita a distribuir blancos hasta lograr la alineación correcta de las palabras que las forman. En este caso, *l* es el acumulador, y se separan las palabras (hasta su marginación) mediante un blanco. La última línea se produce tal cual, sin utilizar la función de marginación. La condición en la instrucción **si** calcula si la palabra "cabe" o no dentro de la línea: si no cabe, se produce la línea (bien formada) y se comienza una nueva línea que consta al principio de una sola palabra; si cabe, basta añadirla.

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

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

lo cual es trivial.

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

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

En el caso de que sea 0, la línea ya está bien ajustada; en otro caso, calculamos el número de espacios en blanco que tiene la línea sin ajustar (esto es, el número 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 sucesión local a la función que proporciona los caracteres de la línea. 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 línea resultante se formará 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;

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

  sucesion 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
    sucesion 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;

  sucesion líneas: línea es
    var p: palabra; l: línea;
  haz   l ← "";
    para p en palabras haz
      si
        □ long(l) + long(p) + 1 > max_línea ⇒ produce margina(l); l ← p;
        □ l = "" ⇒ l ← p;
        □ otros ⇒ l ← l + " " + p;
      fin si;
    fin para;
    produce l;
  fin líneas;

haz
  para l en líneas haz escribe_línea l; fin;
fin programa;
```

Algoritmo 15. Justificador de textos

Ejercicio

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

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

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

Las instrucciones estarán siempre en línea aparte, comenzarán con un punto y constarán de dos letras siguiendo al punto y posiblemente alguna otra indicación. El programa deberá "entender" como mínimo las siguientes:

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

.pa (PÁgina) pasa de página

.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 carácter de cada línea se utilizará para determinar cómo se imprime la línea y no como parte de la línea. Así, una línea que contenga

```
lPrimera página
```

provocará la impresión de "Primera página" ***en la primera linea de la siguiente página***. Los caracteres de control utilizables son:

' ' el espacio en blanco fuerza la impresión de la línea a continuación de la anterior (caso normal).

'1' fuerza la impresión de la línea a principio de la siguente página (salto de página)

'+' fuerza la impresión de la línea ***sobre*** la anterior (sobreimpresión)

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

Tipos Subrango y Enumerados

Tipos enumerados

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

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

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

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

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

Introduciremos formas de definir nuevos tipos de datos análogas a las definiciones de conjuntos por extensión.

Una declaración 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.[10] 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

---

[10] Debe entenderse que T es un *tipo*, no un *conjunto*; y que T sólo puede utilizarse en declaraciones.

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 asignación y la comparación por (no) igualdad.
- **funcion** *ord(x: T): entero;* es lo que se llama el ***ordinal*** de una expresión 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 ***conversión de tipo*** $T(k)$, donde k es una expresión entera, es la función inversa de la *ordORD*: $T(k\text{-}1) = i_k$, $T(ord(i_k)) = i_k$, $ord(T(k)) = k$. Es un error calcular T de $k < 0$ o $k \geq n$.
- Las sucesiones 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 revés con *desc*.
- Puede realizarse entrada y salida con objetos y expresiones de tipo enumerado: se escribirá el identificador que denota el valor deseado.

Si en la declaración del tipo substituimos las llaves por paréntesis

tipo T **es** $(i_1, i_2, \ldots, i_n)$;

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 último, si a la declaración de tipo ordenado le añadimos la palabra reservada **ciclico**

tipo T **es** $(i_1, i_2, \ldots, i_n)$ **ciclico**;

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

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

La ordenación es la misma que en los no cíclicos.

Ejemplos

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

Los tipos no ordenados son el caso más 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 ordenación implícita 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 sucesión *(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 cíclicos, 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 lógico suponer que el sucesor (o siguiente) del *Domingo* es el *Lunes*.

Sintaxis

```
declaración_de_tipo =
  tipo identificador [es [nombre] tipo];

tipo = identificador
  | TIRA "("expresión_constante")"
  | tipo_enumerado
  | tipo_subrango
  | tipo_tabla
  | 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_cíclico

tipo_enumerado_no_ordenado = "{" lista_de_identificadores "}"

tipo_enumerado_ordenado = "(" lista_de_identificadores ")"

tipo_enumerado_cíclico = "(" 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 características con ellos:

- El tipo *entero* podría considerarse declarado como

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

  donde los puntos significan que habría que escribir todos los enteros que admite el ordenador, y ordenados. Desde luego que esto viola la sintaxis; además, la operación &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* podría ser

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

  Incluyendo bien ordenados las letras y los números; 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 contigüidad de los dígitos, esto permite calcular mediante *suc(c)* el cáracter asociado al dígito siguiente al representado en el caracter *c*, y el valor entero que representa un tal dígito *c* mediante

  ord(C) - ord('0')

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

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

  añadiendo además sus propios operadores ("^", "v", "~", "^^", "cor").

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 podría tomar si estuviese declarado con su tipo base, con la limitación de que estos valores deben estar comprendidos entre los valores de las expresiones constantes que lo declaran.

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

Ejemplos

tipo *día* **es** *[1..31];* -- *días del mes; tipo base entero*

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

var *c: caracter; d: dígito;*

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

d ← *c;* -- *también es correcto; si c no está 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 continuación calcula el día siguiente al de una fecha dada, incluyendo día de la semana, día del mes, mes y año. Los conceptos de tipo enumerado y tipo subrango permiten describir con precisión los datos utilizados:

- Los días de la semana se describen adecuadamente mediante

  tipo *día_de_la_semana* **es**
  (Lunes, Martes, Miercoles, Jueves, Viernes, Sábado, 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 días del mes:

  tipo *día* **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 observará que datos erróneos (como 45 de Enero o año -7) son detectados automáticamente como errores gracias a las declaraciones efectuadas. En cuanto a la condición trivial *es_el_último_día_del_mes*, se ha escrito como simplificación, suponiendo que todos los meses tienen 31 días.

```
programa Halla_la_siguiente_fecha es

  tipo día_de_la_semana es
  (Lunes,Martes,Miercoles,Jueves,Viernes,Sábado,Domingo) ciclico;
  tipo mes es
    (Enero,Febrero,Marzo,Abril,Mayo,Junio,
    Julio,Agosto,Septiembre,Octubre,Noviembre,Diciembre) ciclico;
  tipo día 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_último_dia_del_mes haz
    -- para simplificar, suponemos que todos los meses tienen 31 dias; puede pensarse,
    -- como ejercicio, en cuáles son las modificaciones necesarias para tratar
    -- los meses de 30, 29 y 28 días, y la problemática de los años bisiestos
    vale d = 31;
  fin es_el_último_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 día de la semana.
  ds ← suc(ds);

  -- Calculamos la siguiente fecha
  si es_el_último_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 será ",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. Más sobre entrada y salida

```
instrucción =
  instrucción_de_asignación
  | instrucción_de_invocación
  | instrucción_nada
  | instrucción_vale
  | instrucción_acaba
  | instrucción_produce
  | instrucción_sal
  | instrucción_si
  | instrucción_repite
  | instrucción_mientras
  | instrucción_itera
  | instrucción_para
  | instrucción_con
```

Figura 45. Instrucciones

Instrucción mientras

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

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

Definimos la instrucción **mientras**

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

como una abreviación del esquema anterior.

Su forma general es

instrucción_mientras =
mientras *condición* **haz**
instrucción
{*instrucción*}
fin [**mientras**];

Estilo: además del sugerido en la sintaxis,

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

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

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

Instrucciones itera y sal

La forma pura de la iteración se describe mediante la instrucción **itera**:

instrucción_itera
itera
instrucción
{*instrucción*}
fin [**itera**];

Estilo: además del sugerido en la sintaxis,

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

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

que tiene como efecto la ejecución indefinidamente repetida de las instrucciones subordinadas. Suele utilizarse en conjunción con la instrucción **sal**:

```
instrucción_sal = sal [identificador] [si condición];
```

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

en la forma

```
itera
  instruccion(es)
sal si condición;
  instruccion(es)
fin itera;
```

Figura 49. Esquema normal de utilización de la instrucción **itera**

El efecto de la instrucción **sal** es el de terminar la ejecución de la instrucción **itera** y continuar con la primera instrucción que sigue a ésta. Un caso típico de utilización es el siguiente:

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

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

de la cual puede ser la siguiente una utilización típica:

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

La instrucción **sal** es más general: incorpora en su sintaxis la posibilidad de mencionar explícitamente de que iteración se desea salir; aunque es poco frecuente, a veces interesa terminar no con la iteración que inmediatamente engloba a **sal**, sino

con alguna más externa. Esto puede hacerse utilizando un ***nombre de iteración***, en la forma de un identificador seguido de dos puntos que precede a la instrucción cuya ejecución se quiere terminar, y mencionando este identificador en la instrucción **sal**:

```
Iteración_externa:
 itera
   ...
   Iteración_interna:
    itera
      ...
    sal Iteración_externa si C:
      ...
    fin itera;
 fin itera;
 -- la instrucción sal continúa la ejecución en este punto
```

También es posible omitir "**si** condición" en la instrucción **sal**, en cuyo caso se supone "**sal si** *cierto*". De este modo, es equivalente escribir

```
sal si C;
```

y

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

Pueden utilizarse tantas instrucciones **sal** dentro de una iteración como se desee, aunque, por razones de claridad, debe procurarse restringirse al esquema presentado. La instrucción **sal** sirve también para terminar la ejecución de una instrucción **para**, **repite** o **mientras**; por su parte, éstas admiten también un nombre de bucle.

La ejecución de una iteración puede también terminarse mediante la de una instrucción **vale** o **acaba**, o suspenderse con una **produce**.

Algunas equivalencias

La combinación **itera-sal** es más general que las otras formas de iteración: si en la forma

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

suponemos i_1 = **nada;**, obtenemos

itera
  nada;
sal si *C*;
  i_2
fin itera;

que equivale evidentemente a

mientras ~ *C* **haz**
  i_2
fin mientras;

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

itera
  i_1
sal si *C*;
  nada;
fin itera;

que en este caso equivale a

repite
  i_1
hastaque *C*;

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

Forma factorizada de la instrucción si

```
instruccion_si =
  si expresion es
    □ lista_de_valores ⇒
      instruccion
      {instruccion}
    {□ lista_de_valores ⇒
      instruccion
      {instruccion}}
    [□ otros ⇒
      instruccion
      {instruccion}]
  fin [si];

lista_de_valores = valor_o_rango { , valor_o_rango }

valor_o_rango = expresion_constante [ .. expresion_constante ]
```

Estilo: si la(s) instruccion(es) caben en la misma línea que □, puede hacerse

```
  □ lista_de_valores ⇒ instruccion(es);
```

o

```
  □ otros ⇒ instruccion(es);
```

***Figura 50. Sintaxis y estilo de la forma factorizada de la instrucción* si**

Si en una instrucción **si** todas las condiciones se refieren al valor de una misma expresión o variable

```
si
  □ (c = '.') ∨ (c = '*')  ⇒ I1;
  □ (c = 'a') ∨ (c = 'A')  ⇒ I2;
  □ otros ⇒ I3;
fin si;
```

puede abreviarse, sacando la variable *c* como "factor común" y ganando en claridad, por

```
si c es
  □ '.' , '*' ⇒ I1;
  □ 'a' , 'A' ⇒ I2;
  □ otros ⇒ I3;
fin si;
```

haciendo una especie de "factor comun" de la expresión en cuestión. Se observará que los valores posibles de *c* (que antes se enumeraban en las alternativas ("v") de las condiciones, ahora se separan por comas; además, puede utilizarse la notación

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 instrucción **si**, se produce un error si falta "□ **otros**" y la expresión no coincide con ninguno de los valores expresados; se notará que, en esta instrucción, han de ser ***expresiones constantes*** (esto es, expresiones que se evaluen como e involucren sólo constantes), todas ellas distintas. Además, el tipo de la expresión solo puede ser *caracter*, *entero*, *logico*, enumerado o subrango.

La diferencia entre los ordinales del mínimo y el máximo valor del conjunto de las expresiones constantes deberá ser reducida; en caso contrario, se preferirá la forma general de la instrucción **si** (o la forma simplificada): no se escribirá

```
si n es
  □ 1 ⇒ I1;
  □ 100 ⇒ I2;
  □ 1000 ⇒ I3;
fin si;
```

sino

```
si
  □ n = 1 ⇒ I1;
  □ n = 100 ⇒ I2;
  □ n = 1000 ⇒ I3;
fin si;
```

y se preferirá

si
  □ $k \geq 1 \wedge k \leq 100 \Rightarrow I_1$;
  □ $k \geq 101 \wedge k \leq 200 \Rightarrow I_2$;
fin si;

a

si *k* **es**
  □ *1..100* ⇒ I_1;
  □ *101..200* ⇒ I_2;
fin si;

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

si *d* **es**
  □ *lunes..miercoles* ⇒ $instrucción_1$;
  □ *viernes , domingo* ⇒ $instrucción_2$;
  □ **otros** ⇒ $instruccion_3$;
fin si;

Formatos de entrada y salida

Normalmente, las instrucciones *escribe* y *escribe_linea* utilizan automáticamente un determinado número de espacios para escribir cada tipo de datos: el mínimo necesario para todos los tipos, excepto para los reales (como se vera a continuación). Esto quiere decir que una instrucción como

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

para $n = 3$ escribirá

```
Hay 3 elementos.
```

y para $n = 12345$ escribirá

```
Hay 12345 elementos.
```

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

Ejemplo:

escribe ' El número de As es: ',n:3;

para $n = 5$ produce

```
El número de As es:    5
```

Si el número de espacios especificados no es suficiente para acomodar el valor a escribir, automáticamente se toman más espacios hasta poder escribir la totalidad del valor: la instrucción anterior, con $n = 1234$, produciría

```
El número de As es: 1234
```

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

El caso de los números reales es algo más complejo: se considera que la expresión mínima de un número real consta de

un espacio en blanco
el signo (opcional) del número
un dígito antes del punto decimal
el punto decimal
un dígito después del punto decimal
la E de exponente
el signo del exponente
dos dígitos 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 número utilizando su expresión mínima. A medida que el formato especifica un número mayor de espacios, crece el número de dígitos que aparecen después del punto decimal, hasta llegar al máximo de dígitos que permite la versión. El espacio adicional que especifique el formato se utiliza en forma de espacios en blanco que preceden al número.

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 diálogos interactivos entre el programa y el usuario de modos no obvios. La transmisión de datos sobre la pantalla se realiza línea a línea: esto quiere decir que varias instrucciones *escribe* "acumulan" sus resultados hasta la ejecución de una *escribe_linea*, y que varias *lee* utilizan la misma línea de entrada, si hay datos y no media alguna *lee_linea*. Por ejemplo, si programamos

```
lee m,n;
```

y la línea de entrada contiene

```
2 4 6
```

M y *n* tomarán por valor, respectivamente, 2 y 4, pero el número 6 quedará pendiente de ser leido; una instrucción siguiente

```
lee p;
```

asignara 6 a *p*. Para evitar esta acumulación, se utiliza a veces la instrucción *lee_linea*; pero si escribimos en vez de *lee*

```
lee_linea m,n;
```

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

```
lee_linea; lee p;
```

que ignorará el resto de la línea anterior, y obtendrá de la nueva el dato que buscamos.

En general, el esquema de diálogo 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 diálogo interactivo

Cálculo del Máximo Común Divisor

El siguiente programa calcula el Máximo Común Divisor de dos números naturales, basándose en las reglas

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

y es interesante como ejemplo de las instrucciones explicadas.

```
programa Máximo_Común_Divisor es
  var a,b: entero;
haz
  itera
    itera
      escribe_linea "Escribe dos números enteros, o 0 0 para terminar:";
      lee a,b;
    sal si (a > 0 ∧ b > 0) ∨ (a = 0 ∧ b = 0);
      escribe_linea "Han de ser positivos.";
    fin itera;
  sal si a = 0 ∧ 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. Cálculo del Máximo Común Divisor

Ejercicio

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

Conjuntos

Los conjuntos como abreviación de expresiones

Una condición como

(C = 'a') ∨ (c = 'e') ∨ (c = 'i') ∨ (c = 'e') ∨ (c = 'u')

puede cambiarse, con el mismo efecto, por

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

utilizando un ***conjunto***[11] y el ***operador de pertenencia*** "**en**". Pueden escribirse conjuntos con elementos de tipo *entero*, *caracter*, *logico*, enumerado o subrango (con las limitaciones que se estudiarán a continuación) encerrando sus elementos entre llaves; los elementos pueden ser (expresiones) constantes, variables o expresiones, y el significado es el usual en matemáticas elementales.

conjunto = "{" expresión {, expresión} "}"

Figura 52. Sintaxis de un conjunto

Tipos conjunto y operaciones

También pueden definirse objetos que contengan tales conjuntos, directamente, o mediante la definición de un tipo

[11] No debe confundirse con un tipo enumerado no ordenado.

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 éstos o de *entero*; sus ordinales mínimo y máximo están sujetos a limitaciones dependientes de versión (generalmente, el mínimo no puede ser menor que cero, y el máximo que el mayor ordinal de un carácter).

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 deberán ser (expresiones) constantes:

const *mol* = {*sofa,cama*};

Las operaciones aplicables a los conjuntos son:

- La asignación y la comparación 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 sólo si el elemento pertenece al conjunto.
- La ***inclusion*** conjuntista (no estricta), que permite determinar si un conjunto está contenido en otro o, lo que es lo mismo, si cada elemento de un conjunto pertenece tambien a otro.

  $A \leq B \Leftrightarrow \sim$ **existe** x **en** A **talque** $\sim (x$ **en** $B)$
  $A \geq B \Leftrightarrow \sim$ **existe** x **en** B **talque** $\sim (x$ **en** $A)$

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

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

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

A incluido estrictamente en $B \Leftrightarrow A \leq B \wedge A \neq B$

y simétricamente.

- La ***unión*** (operador "+"), ***intersección*** (operador "*") y ***diferencia*** (operador "-") de conjuntos, con el significado matemático tradicional:

$A + B = \{x \mid x \textbf{ en } A \vee x \textbf{ en } B\}$
$A * B = \{x \mid x \textbf{ en } A \wedge x \textbf{ en } B\}$
$A - B = \{x \mid x \textbf{ en } A \wedge \sim (x \textbf{ en } B)\}$

Por ejemplo:

{mesa,silla} + {sofá} = {mesa,silla,sofá} -- unión

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

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

{1,2,3,4,5} - {2,4,6,8} = {1,3,5} -- diferencia:
-- elementos del primer conjunto que no están 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 vacío***, que tiene cualquier tipo (conjunto), y se denota {}.

Algunos ejemplos

Para determinar si un carácter es una letra mayúscula, puede ser muy útil una constante

```
const mayúsculas =
  {'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 permitirá preguntas del tipo

si *c* **en** *mayúsculas* **entonces**....

Generación de números primos mediante la criba de Eratóstenes y conjuntos: Un método bastante conocido para generar una lista de números primos es el siguiente: se parte de la lista de los números 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*

Después de tachar el número 1 (que no es primo), se mira el primer número no tachado (2). Este número es primo (se apunta en la lista); se tachan todos los múltiplos 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 número es el 3 (primo, que se apunta en la lista). Se tachan ahora los múltiplos 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 número (5) es primo ... Se repite así 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 observará que, dada la ausencia de una operación predefinida para calcular complementos, es necesario construir el conjunto de los naturales "a mano" (instrucción **para**).

```
programa Números_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 ∧ entonces ~ (p en T) haz
      p ← p + 1;
    fin mientras;
    -- (sólo 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 múltiplos 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. Generación de números primos la Criba de Eratóstenes

Tablas

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 más complicado:

```
si
  □ (n1 ≤ n2) ∧ (n1 ≤ n3)  ⇒  min ← n1;
  □ (n2 ≤ n1) ∧ (n2 ≤ n3)  ⇒  min ← n2;
  □ (n3 ≤ n1) ∧ (n3 ≤ n2)  ⇒  min ← n3;
fin si;
```

creciendo la complejidad con el número de variables. Si de algún modo pudiese hablarse de n_i, siendo i una *variable* (o una expresión) entre 1 y k, se podría 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 a que no es posible escribir subíndices y a la ambigüedad que resultaría de intentar diferenciar n_i y el identificador *ni*). Aunque con otra notación, 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 expresión como ***índice***:

n[1] -- *antes* n_1
n[3] -- n_3
n[i] -- n_i
n[i + j] -- *el índice es una expresión*
n[n[1]] -- *por qué no?*

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

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 subindicadas***

Naturalmente, también se podría haber utilizado un tipo

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

que hubiese permitido declarar varios objetos similares.

Aplicaciones

Para pasar un carácter "a mayúsculas" (es decir, transformar los caracteres que sean minúsculas en sus correspondientes mayúsculas) necesitamos algún mecanismo para implementar la aplicación

Mayúscula

'a' ——————> *'A'*
'b' ——————> *'B'*
'c' ——————> *'C'*
'd' ——————> *'D'*
. . . .
'z' ——————> *'Z'*

Además del método obvio pero pesado de utilizar una función

```
funcion mayúscula(c: caracter): caracter haz
  si c es
    □ 'a' ⇒ vale 'A';
    ...
    □ 'z' ⇒ vale 'Z';
  fin si;
fin mayúscula;
```

o del más 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 **tabla**:

se declara

```
var mayuscula: tabla caracter de caracter;
```

y se inicializa

```
mayúscula['a'] ← 'A';
mayúscula['b'] ← 'B';
mayúscula['c'] ← 'C';
  . . .
mayúscula['z'] ← 'Z';
  . . .
c ← mayúscula[c];
```

que reduce el cálculo al contener la aplicación como objeto siempre calculado en memoria.

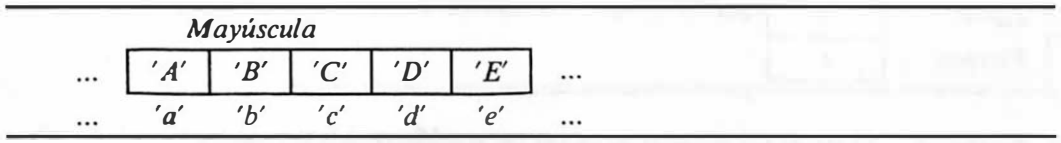


***Figura 55. Fragmento de una aplicación de los caracteres en los caracteres representada mediante una* tabla**

Tabulación

En un determinado trabajo, cada día

tipo *dia* **es** *(Lunes,Martes,Miercoles,Jueves,Viernes,Sábado,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 expresión

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 días 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;
```

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;
```

El tipo tabla

```
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**

Un objeto de tipo **tabla** representa una ***familia indexada*** de objetos, que son accesibles individualmente mediante la operación de ***selección***.

```
variable = identificador {selector}

selector = ↑
  | "[" expresión {,expresión} "]"
  | . identificador
```

Figura 58. Sintaxis de una variable con sus posibles selectores: "↑" y ". identificador" se estudiarán mas adelante

Hay tantos ***elementos*** o ***componentes*** como valores en el *tipo_indice*, que ha de ser *caracter*, *logico*, enumerado o subrango; cada elemento se selecciona añadiendo al nombre del objeto el valor del índice 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

igualmente,

v[il][i2]...[in]

es intercambiable por

v[il,i2,...,in]

con el mismo sentido.

Operaciones

Las operaciones aplicables a los objetos de estos tipos son:

- La asignación.
- La comparación por igualdad o por desigualdad.
- La ***selección***. Esta es una operación 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 asignárseles 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 números, escribir sólo los que superen la media de todos ellos."

Y admite varias interpretaciones. Algunas de las posibles son: la cantidad de números es fija, o variable; y, en ese caso, viene determinada por un marcador (p. ej., el número 0) al final de la serie, o indicada al principio en un número adicional.

Cantidad fija de números: por ejemplo, 10 números. El problema no puede resolverse con la aproximación utilizada en casos anteriores, ya que necesitamos "pasar dos veces" por cada número: la primera, para calcular la media, y la segunda, para ver si la excede. Utilizaremos una **tabla** para almacenar los números.

```
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 así, si el enunciado cambia hacia otra cantidad distinta de 10, basta con cambiar la constante. Por lo demás, el programa es trivial.

```
programa Mayores_que_su_media_versión_1 es
  const max = 10; -- cantidad de números
  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 números los mayores que su media

Cantidad variable de números: En este caso, el mecanismo que suele utilizarse es suponer un límite superior razonable (p. ej., 100) a la cantidad de números, y declarar una tabla de ese tamaño:

```
const max = 100;
var N: tabla [1..max] de entero;
```

También se declara una variable para contener la cantidad

```
var c: entero;
```

y se trabaja como si N estuviera declarado con un índice igual a $[1..c]$ (cosa que el lenguaje no permite directamente, ya que los límites de un tipo subrango deben ser constantes). Con ello se consigue el efecto de ***dimensiones ajustables***.

Si la serie de números está terminada por un cero, el programa será

```
programa Mayores_que_su_media_versión_2 es
  const max = 10; -- máximo de números
  var N: tabla [1..max] de entero;
  var c, i, media: entero;
haz
  c ← 1; -- cantidad de números 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 si 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 números terminada por 0 los mayores que su media***

Y si el primer número no pertenece a la serie, sino que indica cuantos le siguen,

```
programa Mayores_que_su_media_versión_3 es
  const max = 10; -- máxima cantidad de números
  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 números los mayores que su media***

En este caso el esquema es prácticamente igual al de la Versión 1.

Inversión de una serie de números

Disponemos de una serie de números 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 límite superior a la longitud de la serie, y utilizamos una **tabla** como depósito lugar de la inversión, con ayuda de una variable que hace de señal o ***apuntador***:

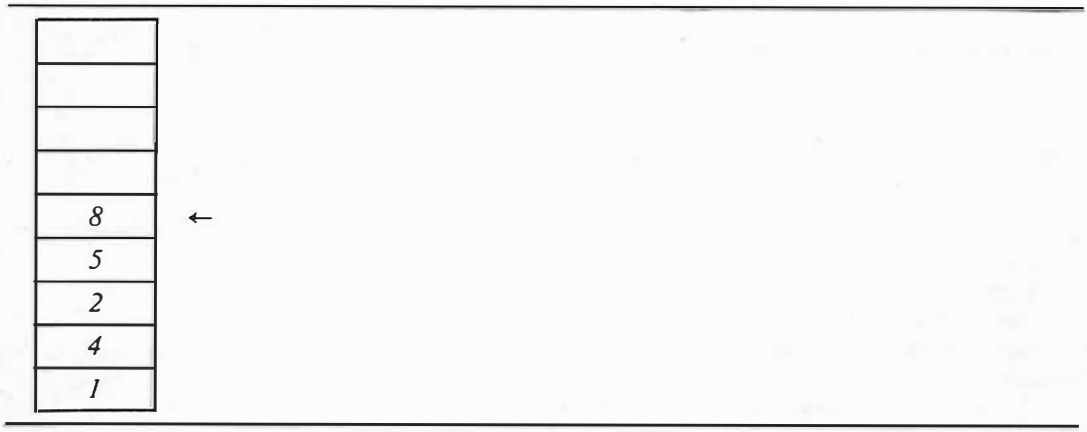


Figura 59. Utilización combinada de una* tabla *y un apuntador

La señal, representada por una flecha en la figura, es una variable cuyo valor es el índice del último elemento leído. 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 *desc*endente 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, ningún número
  itera
    lee n;
  sal si 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 números

Matrices

Así como a las tablas se les llama a veces ***vectores***, las tablas de tablas 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 denotará por *M[i,j]* o *M[i][j]*.

Las matrices pueden ser bidimensionales, tridimensionales o, en general, de cuaquier número de dimensiones, según las necesidades del programa. Si se utilizan menos subíndices de los posibles (si en el ejemplo anterior hablamos de *M[1]*) se obtiene una ***submatriz*** de la matriz total, que generalmente representa una fila o columna de la abstracción dada.

var *M:* **tabla** *[1..3],[1..4]* **de** *entero;*

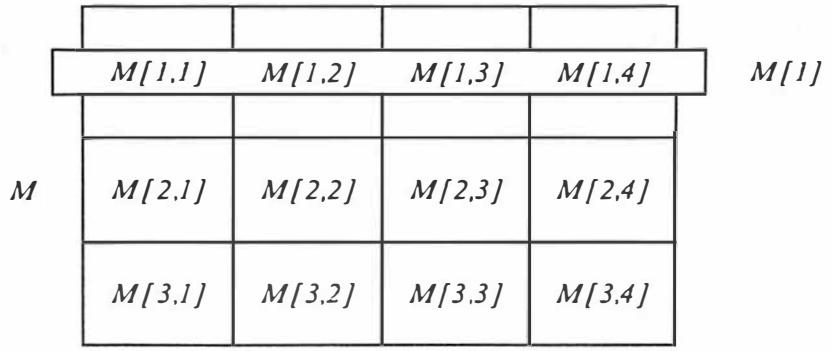


Figura 60. ***Una matriz con una submatriz:*** *M* es la matriz completa. El rectángulo que sobresale delimita la submatriz *M[1]*, que a su vez es una **tabla**, con componentes *M[1][1]* = *M[1,1]*, *M[1][2]*, *M[1][3]* y *M[1][4]*, y tiene tipo **tabla** *[1..4]* **de** *entero*.

Un ejemplo de tratamiento de matrices: generación de Cuadrados Mágicos

Llamaremos cuadrado mágico a una matriz cuadrada de números, 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 61. ***Ejemplo de Cuadrado Mágico de lado 5***

El siguiente programa, dado un número entero impar *n*, construye e imprime un cuadrado mágico de tamaño *n*n*. Para ello, utiliza las siguientes reglas:

1. En todo momento se supone que los bordes del cuadrado están identificados. Dicho de otro modo, convenimos en hablar como si, por ejemplo, a la última casilla de la primera fila le siguiera la primera casilla de la misma fila por la derecha, y la última de la última fila por arriba; y así para cada casilla. Con ello conseguimos que toda casilla tenga "vecinos" en cualquier dirección.

2. Para llenar el cuadrado, se escriben sucesivamente los *n*n* primeros números 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 dirección diagonal descendente.

El programa es también un buen ejemplo de diseño descendente y de escritura con formatos.

```
programa Cuadrado_mágico es

  const max = 25; -- tamaño máximo del cuadrado mágico

  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 dimensión del cuadrado:";
    itera
      lee n;
    sal si n ≥ 3 ∧ n ≤ max ∧ impar(n);
      escribe_linea "Valor erróneo. 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_número 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 está_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 está_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_número;
    siguiente_casilla;
    si está_ocupada entonces rebota; fin;
  hastaque está_lleno;
  imprime_resultados;
fin programa;
```

Algoritmo 23. Generación de Cuadrados Mágicos.

Generación de una tabla de números primos, con una discusión sobre optimización de programas

En el Algoritmo 14 en la página 90 se estudia una condición para determinar si un número es o no primo. Una ampliación de ese esquema, utilizando una

```
T: tabla [1..n] de entero;
```

para contener los primos (*T[i]* es el *i*-ésimo primo), es la siguiente:

```
T[1] ← 1; T[2] ← 2; T[3] ← 3; -- los primeros primos
l ← 3; -- índice o lugar del último primo calculado
mientras l < n haz
  si existe j en asc(T[1] + 2,infinito) talque
    ~ existe k en asc(2,1) 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 número a partir del último primo mas 2 (mas 1 no tiene interés, ya que es un número 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 construcción están en la tabla *T*).

Una vez encontrado un número primo, se mete en la tabla. Para ello, se dispone de una variable *l* entera, que por convención siempre "marca" el índice del último primo calculado; al encontrar uno nuevo, se incrementa ese índice, y se asigna a *T[l]* su valor, preservando así la convención sobre *l*. La iteración se encarga de llenar la tabla de primos (hasta la constante *n*, tamaño de la tabla).

Un análisis detallado del programa muestra que se realizan bastantes cálculos inútiles al verificar si un candidato a primo lo es o no, pues se halla su módulo respecto de todos los primos anteriores, mientras que sólo es necesario hallarlo respecto de los primos que no excedan a su raíz cuadrada

> En efecto, si un candidato *j* es divisible exactamente por un primo *k* mayor que su raíz cuadrada, el cociente *j/k* es un divisor de *j* menor que su raíz cuadrada. *J/k* contendra algun factor primo (llamemosle *q*), también menor que *raiz(j)*; por todo lo cual, de ser *k* divisor de *j*, también lo es *q*, primo menor que *k*, que por construcción ya se ha analizado antes como divisor. De donde resulta que el cálculo es supérfluo.

Cambiaremos el segundo existencial por

~ **existe** *q* **en** *divisores* **talque** *j* **mod** *q* = *0*

definiendo la **sucesion**

```
sucesion divisores: entero haz
  k ← 2;
  repite
    produce T[k];
    k ← k + 1;
  hastaque T[k] * T[k] > j;
fin divisores;
```

También puede observarse que *j* toma muchos valores que es supérfluo 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*). Podría también pensarse en evitar los múltiplos de 3: bastaría 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 optimización del tiempo de proceso.

El programa final será:

```
programa Números_primos es
  const n = 200; -- cantidad de números primos
  const infinito = 2000000000; -- más o menos
  var p,q,j,k,l: entero;
  var T: tabla [1..n] de entero; -- los primos

  sucesion 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," números primos.";
  escribe_linea 1:5;
  escribe_linea 2:5;
  escribe_linea 3:5;
  l ← 3; -- índice o lugar del último primo calculado
  mientras l < n haz
    si existe j en asc(T[l]+2,infinito) talque
      ~ 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. Generación de números primos

El problema de la siguiente permutación

Supuesto un conjunto C ordenado, p. ej. un conjunto de dígitos

$C = \{0,1,2\}$

se llama ***permutación en C*** a cualquier ordenación 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 Permutación se plantea como sigue: dada una permutación, hallar la siguiente en orden *alfabético*, esto es, siguiendo el mismo criterio mediante el cual definimos una ordenación entre las tiras de caracteres de igual longitud. Se notará que este criterio, aplicado a permutaciones numéricas, equivale a la ordenación numérica habitual; en el caso de caracteres, significa que ABC es menor que BAC, lo cual es lógico. "La siguiente" significa la menor que supera a la dada.

Para determinar cual es la siguiente permutación a una dada, realizaremos el siguiente análisis:

- En primer lugar, y supuesto el problema resuelto, es obvio que, entre una permutación y la siguiente, habrá una sección inicial (posiblemente vacía) 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 permutación
  035124 -- la siguiente
  '03' es invariable.
  El resto cambia.
  El primer elemento en que no coinciden está en la posición tercera: pasa de '4' a '5'.

  y afirmamos que este elemento, principio de la parte derecha de la permutación en que las dos diferirán, puede hallarse como el primer número de la permutación original empezando por la derecha y yendo hacia atrás en el que la sucesión deja de ser creciente (en el ejemplo presentado, a partir de la primera permutación, encontramos la serie '1', '2', '5', '4', y es '4' el primero que rompe la sucesión ascendente)

  > Debido a que, de estar situado más a la derecha, la parte que debería cambiar no podría hacerlo hacia una formación mayor
  >
  > > Si intentamos cambiar a partir del 1, el 2 o el 5, es imposible encontrar una formación mayor que la dada.
  >
  > por estar ésta ordenada decrecientemente, lo que le asigna la mejor posición en la ordenación; y de estarlo más a la izquierda, forzosamente se realizaría un cambio mayor que el anunciado
  >
  > > No hay ninguna manera de, cambiando a partir del '0' o el '3', obtener una permutación 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 número de los que le siguen que le supera

  > Ya que es la única forma de obtener la menor permutación

  y que el resto debe ordenarse en forma ascendente (que es justamente la mínima 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 sucesión que produce permutaciones, de tipo

tipo *permutación* **es tabla** *[1..10]* **de** *[0..9];*

donde una variable *n* indicará cuántos de los elementos de la permutación se utilizan (p. ej., en el último ejemplo se utilizan 6 elementos).

El esquema de la sucesión es el siguiente:

```
produce p; -- permutación inicial
mientras existe i en desc(n-1,1) talque P[i] < P[i+1]
  ^^ 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 identificación del lugar de los dígitos que deben intercambarse. El resto es trivial en diseño descendente.

```
programa permutaciones es

  var n: entero;  -- permutaciones de los 'n' primeros números desde 0
  tipo permutación es tabla [1..10] de [0..9];  -- 1 ≤ n ≤ 10
  var P: permutación;

  sucesion permutaciones: permutación 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; -- permutación inicial
    mientras existe i en desc(n-1,1) talque P[i] < P[i+1]
      ^^ 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_permutación es var i: entero; haz
    para i en asc(1,n) haz escribe P[i]; fin;
    escribe_linea;
  fin escribe_permutación;

  accion inicializa es var i: entero; haz
    itera
      escribe_linea "De cuántos elementos quieres realizar permutaciones?"; lee n;
    sal si n > 1 ^ 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_permutación; fin;
fin programa;
```

Algoritmo 25. Generación de permutaciones de N elementos por el método de la Siguiente Permutación

Manipulación de Matrices

El siguiente programa define acciones y funciones para la manipulación 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 sí se limita a demostrar la utilización de esos subprogramas. Se utilizan parámetros por constante para evitar copias innecesarias de datos.

```
programa manejo_de_matrices es

  const dim = 3; -- dimensión 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,
    " números reales para dar valor a la primera matriz:";
  lee_matriz A;

  escribe_linea "Escribe ",dim*dim,
    " números 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 sí. Interesa hallar todas las formas posibles de hacerlo.

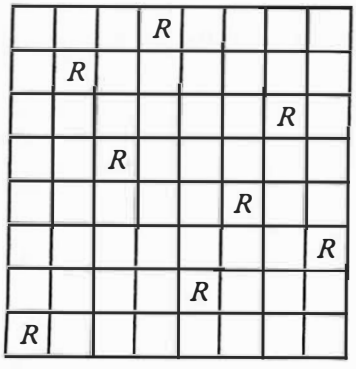


Figura 62. Una de las soluciones al problema de las Ocho Reinas

Una primera aproximación a la resolución del problema podría 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 sí).

```
para reina1 en posiciones_del_tablero haz
  para reina2 en posiciones_del_tablero haz
    ...
      para reina8 en posiciones_del_tablero haz
        si ~ se_matan entonces solución;
        sino ...
        fin si;
      fin para;
    ...
  fin para;
fin para;
```

Si estimamos el número de combinaciones que habría 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 reducción notable del número de combinaciones si se considera que, dado que dos reinas que se hallen en la misma fila se matan entre sí, no se pierden soluciones si convenimos en asignar una reina a cada una de las filas, permitiéndole moverse sólo dentro de esa fila.

```
para reina1 en fila1 haz
  para reina2 en fila2 haz
    ...
      para reina8 en fila8 haz
        si ~ se_matan entonces solución;
        sino ...
        fin; si;
      fin para;
    ...
  fin para;
fin para;
```

El número 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 también excesivo, aunque ya es manejable por un ordenador grande.

Un análisis más fino del problema permite ver que una vez colocadas algunas de las reinas, no tiene sentido probar de colocar las demás en lugares en los que se maten con las anteriores; y sugiere un programa que vaya "probando" las distintas posibilidades de colocación, y deseche inmediatamente las que provoquen conflictos (el método es más general, se aplica como técnica de construcción de algoritmos, y se conoce como ***backtracking*** o ***retroceso***).

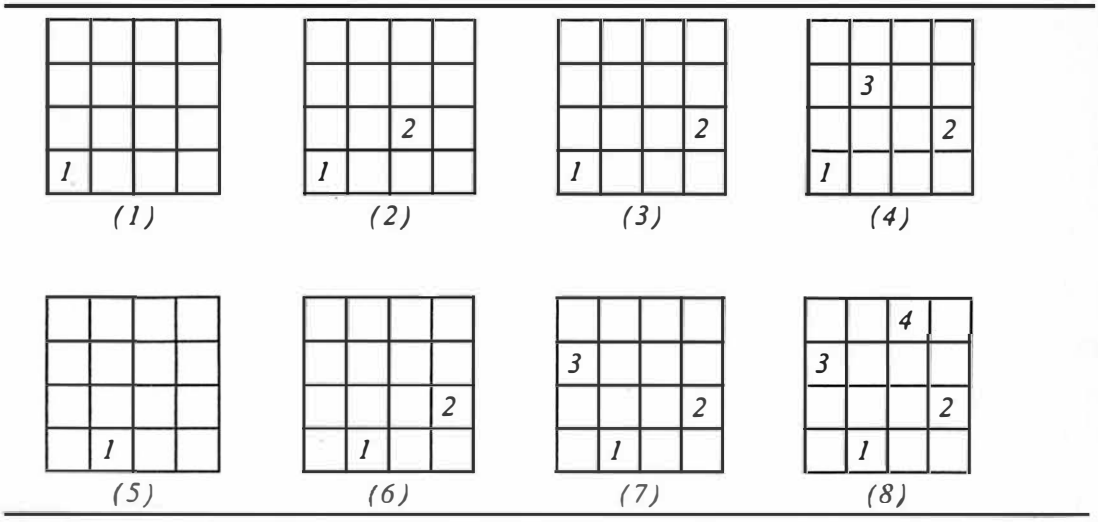


Figura 63. ***Fragmento del proceso de Backtracking para las Ocho Reinas en un tablero de 4x4:*** (1) representa el primer paso del proceso: la reina 1 se pone en la primera casilla de la fila 1. En (2), se ha colocado la reina 2 en la primera casilla de la fila 2 en la que no se mata con otra (ya que 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) [corrección]. En (4) se coloca la reina 3 en la unica posición 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 también la 2, que ya no puede ponerse en otro sitio. La posición 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 solución al problema (8).

El programa está basado en el algoritmo descrito. En cada paso se controla que columnas y diagonales están "ocupadas", en el sentido de que no es posible poner ahi una reina porque se mataría con una ya puesta. La numeración de las columnas es la usual, de 1 a 8; las diagonales se numeran basándose 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: tabla [-7..7] de logico;
var dd: tabla [2..16] de logico;
var col: tabla [1..8] de logico;
```

permiten el control descrito.

Del tablero sólo nos interesa la posición 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 número 1 debe eliminarse).

El resto del programa está tomado directamente de la descripción de la Figura 63 en la página 148.

```
programa Ocho_reinas es

  var f:[0..8]; acabo_de_poner_reina: logico;
  var da: tabla [-7..7] de logico; dd: tabla [2..16] de logico;
  var col: tabla [1..8] de 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 solución es var i: entero; haz
    para i en asc(1,8) haz escribe ' ',D[i]; fin; escribe_linea;
  fin solución;

  accion saca_reina haz
    si candidato < 9 entonces
      da[f-D[f]] ← falso; dd[f+D[f]] ← falso; col[D[f]] ← falso;
    fin si;
    f ← f - 1; acabo_de_poner_reina ← falso;
  fin saca_reina;

  condicion puedo_poner_otra haz
    candidato← 1; f ← f + 1;
    repite
      si ~ col[candidato] ∧ ~ da[f-candidato] ∧ ~ dd[f+candidato]
        entonces vale cierto;
      fin si;
      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 ~ col[candidato] ∧ ~ da[f-candidato] ∧
        ~ 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
      si
        □ hay_8 ⇒ solución; saca_reina;
        □ puedo_poner_otra ⇒ pon_reina;
        □ otros ⇒ acabo_de_poner_reina ← falso;
      fin si;
    sino si puedo_moverla entonces pon_reina; sino saca_reina; fin;
    fin si;
  hastaque ~ quedan_reinas;
fin programa;
```

Algoritmo 27. Las Ocho Reinas

Algoritmos de ordenación

Sea *I* un tipo ordenado apto para ser índice de una **tabla**, con límites *min* y *max*:

```
tipo I es [min..max];
```

y *O* un tipo provisto de todas las operaciones de comparación. Se dice que un objeto

tipo *vector* **es tabla** *I* **de** *O;*
var *T: vector;*

está ***ordenado*** si, al recorrer la tabla, los elementos *T[j]* se encuentran en orden creciente; o, más formalmente,

{1} T está 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 continuación algunos algoritmos de ordenacion.

Aproximación inicial: La fórmula {1} puede ser expresada con más rigor como

{1b} T está ordenado ⇔ **para** *j* **en** *I* **para** *k* **en** *I* **talque** *k* > *j: T[j]* < *T[k]*

y de esta formulación 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. Ordenación de vectores, método 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 mínimo, llevar el menor elemento al principio del vector; a continuación, se repite el proceso con los demás elementos.

Método 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 están mal ordenados.

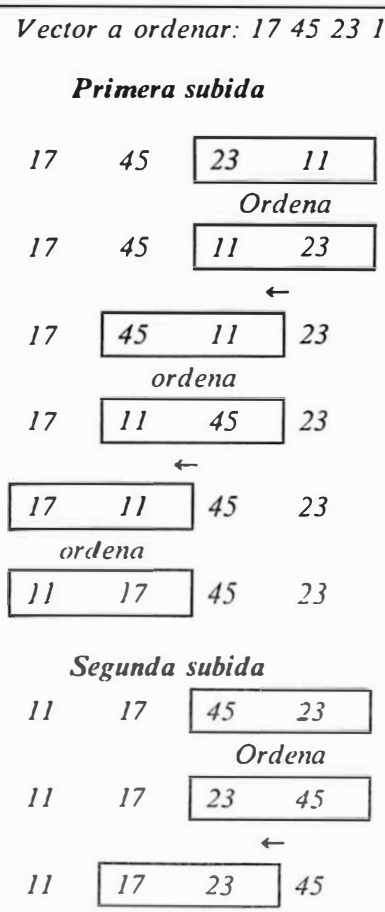


Figura 64. ***Ordenación por el método de la burbuja:*** La burbuja se representa mediante una cajita; se supone que "arriba" es a la izquierda, y "abajo" la derecha. En este caso sólo se necesita hacerla subir dos veces. Cada vez que pasa, ordena si es necesario.

Cada vez que la burbuja sube, sólo necesita hacerlo hasta un lugar menos que la vez anterior, ya que el menor elemento es llevado con seguridad a la primera posición (la más alta) por la burbuja en cada vuelta.

El algoritmo resulta directamente de la explicación 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. Ordenación de vectores por el método de la burbuja

Búsqueda de elementos en una tabla

Otro problema que también se presenta con frecuencia es el de la ***búsqueda*** de un elemento que tome determinado valor x (con más propiedad, lo que se busca es su subíndice). En el caso de un vector no ordenado no hay más remedio que realizar una inspección exhaustiva:

```
si existe i en asc(min,max) talque T[i] = x entonces
  el_subíndice_es I;
sino no_hay_un_tal_subíndice;
fin si;
```

Si el vector está ordenado, un método mucho más rápido es el de ***búsqueda binaria***, basado en la misma idea que el método de bipartición presentado en el Algoritmo 13 en la página 84. Suponemos que el tipo índice *I* es numérico para simplificar (sino, se podría recurrir a conversiones de tipo), y presentamos el algoritmo en forma de **accion** con dos parámetros: el segundo, de tipo *logico*, indica si existe el valor en la tabla o no, y el primero contiene su índice si el segundo vale *cierto*.

```
accion búsqueda_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 ∨ j > k;
  indice ← m;
  encontrado ← j ≤ k;
fin búsqueda_binaria;
```

Algoritmo 30. Búsqueda binaria

Ejercicios

1. Dada una lista de números terminada por un cero, escribirla ordenada, primero en orden creciente, y luego en orden decreciente.

2. Escribir un conjunto de subprogramas análogos a los del Algoritmo 26 en la página 145 para tratar vectores en el espacio ordinario, incluyendo suma, resta, producto escalar, producto por un escalar y norma.

3. Hacer un programa que evalúe polinomios: los coeficientes se introducen como serie terminada por un cero; a continuación van valores de x, también en forma de serie con cero, para cada uno de los cuales se evalúa el polinomio. Una serie vacía termina la ejecución 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 información suplementaria. Las minúsculas indican mayúsculas "superralladas". Escribir un programa que convierta un número entero a su representación romana, y viceversa.

5. Dado un texto en el que intervienen no más de 100 palabras distintas, escribir una tabla de frecuencias de aparición de esas palabras.

6. Extender el Algoritmo 27 en la página 150 para que sólo imprima las soluciones esencialmente distintas, es decir, las que no sean giros, simetrías o inversiones una respecto de otra.

7. Dado un número 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:

$$A \times B = \{(a,b) \mid a \text{ en } A \wedge b \text{ en } B\}$$

Dado un par (a,b), se llama ***proyección sobre*** A o ***primera proyección*** al elemento a, y proyección sobre B o segunda proyección al elemento b; a y b son las ***componentes*** del par.

El concepto de producto cartesiano se extiende fácilmente para productos de varios conjuntos, e igualmente sucede con las proyecciones.

Uniones Disjuntas

Dados dos conjuntos A y B, se llama ***unión disjunta*** de A y B al conjunto

$$A \oplus B = \{(a,\{A\}), a \text{ en } A\} \cup \{(b,\{B\}), b \text{ en } B\} = (A \times \{A\}) \cup (B \times \{B\})$$

Alternativamente: la unión disjunta de A y B contiene los elementos de A distinguidos con el subíndice "A" y los de B con el subíndice "B".

Para conjuntos A y B disjuntos, el resultado es equivalente a la unión de A y B, pero para conjuntos no disjuntos, el subíndice permite diferenciar el conjunto de origen de cada elemento:

La unión de los naturales ($\boldsymbol{N}$) y los reales ($\boldsymbol{R}$) es $\boldsymbol{R}$, pero su unión disjunta es

$$\{(a,b) \mid (a \text{ en } \boldsymbol{N} \wedge b = \{\boldsymbol{N}\}) \vee (a \text{ en } \boldsymbol{R} \wedge b = \{\boldsymbol{R}\})\}$$

Esta definición se generaliza inmediatamente a uniones disjuntas de varios conjuntos.

Creación de un tipo para representar números complejos

A partir del tipo predefinido *real*, deseamos construir un tipo *complejo* cuyo conjunto de valores permita representar los números complejos. Dado que el conjunto ***C*** de los números complejos puede estudiarse como producto cartesiano del conjunto de los reales (***R***) por sí mismo, ***R*** × ***R***, estamos interesados en construir un tipo cuyos valores sean ***pares ordenados*** o ***2-tuplas*** de reales. Los matemáticos hablan, dado un valor de ***R*** × ***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 números para distinguirlos: en este ejemplo, llamaremos ***re*** a la parte real (primera componente) e *im* a la parte imaginaria (segunda componente) del número 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 definición, podremos declarar objetos de tipo complejo

var *c,c1,c2: complejo;*

y realizar operaciones de asignación o comparación de (des)igualdad entre ellos:

c ← *c1;*

si *c2* ≠ *c1* **entonces** ...

No es posible comparar tuplas mediante los operadores <, ≤, >, ≥, ya que no existe una ordenación natural o canónica 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 notación

c.re y *c.im*

La operación de seleccionar un campo entre los que componen un objeto de tipo tupla es un modo de ***selección***. En nuestro caso, estos campos tienen tipo ***real***, y pueden ser utilizados en cualquier ocasión en la que se permita una ***variable*** real; en

particular, es posible realizar asignaciones, comparaciones (sin restricción), operaciones aritméticas, etc.

para asignar el complejo *(2,3)* al objeto *c*, utilizaremos

c.re ← *2; c.im* ← *3;*

Para poder trabajar con la noción de suma de números 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 números 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 ***homogéneas***, 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 ***heterogéneos***, ya que cada campo puede tener su propio tipo, posiblemente distinto de los demás tipos que intervienen en la **tupla**.

```
tipo día 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: día; m: mes; a: año; fin;

var f1, f2: fecha;

-- Los siguientes pueden ser valores de F1 y F2 durante la ejecución 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 vv
    (f1.a = f2.a ^^
      (f1.m < f2.m vv
        (f1.m = f2.m ^^ f1.d < f2.d)));
fin menor;
```

Figura 65. 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: además del sugerido en la sintaxis,

```
tupla campo(s): tipo; {campo(s): tipo} fin;
```

***Figura 66. Sintaxis y estilo de un tipo* tupla**

Los identificadores de los campos pueden ser idénticos a otros, declarados en el programa o predefinidos, aunque han de ser distintos entre sí; dado que (excepto con el uso de la instrucción **con**, que se estudiará más adelante en este mismo capítulo) el único modo de referirse a un campo es mediante la operación de selección, no hay ambigüedad posible en la identificación.

```
var x: real;
var z: tupla x: entero; fin;
```

No hay peligro de ambigüedad, 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 asignación, la comparación por (des)igualdad, y la ***selección***:

variable_tupla.campo

se refiere al campo con nombre *campo* del objeto *variable_tupla*. Los campos actúan 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), así como se permiten tablas de tablas, tuplas de tablas, o cualquier combinación:

```
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 **tupla**s se utilizan a menudo para diseñar tipos cuyos valores agrupan información 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 información es opcional (en una ficha: sólo debe ser llenada si se cumplen algunos requisitos en los campos anteriores [pues de otro modo no tendría sentido]; en UBL: algunos campos deben existir sólo si otros campos tienen determinado valor). Las **tupla**s con variantes permiten diseñar tipos de datos apropiados a esas estructuras.

```
tupla_con_variantes =
  tupla
    {campo {,campo}: tipo;}
    parte_variante
  fin [tupla];

parte_variante =
  si discriminante: tipo_del_discriminante es
    □ lista_de_valores ⇒
      {campo {,campo}: tipo;}
      [parte_variante]
    {□ lista_de_valores ⇒
      {campo {,campo}: tipo;}}
      [parte_variante]
    [□ otros ⇒
      {campo {,campo}: tipo;}]
      [parte_variante]
  fin [si];

lista_de_valores = valor_o_rango { , valor_o_rango }

valor_o_rango = expresión_constante [ .. expresión_constante]

discriminante = identificador

tipo_del_discriminante = identificador
```

Figura 67. ***Sintaxis de una*** **tupla** ***con variantes:*** Nótese que la sintaxis de una *parte_variante* especifica que ésta puede contener a su vez otras *parte_variantes*

Llamaremos ***campos fijos*** a los que aparecen antes de **si**, y ***campos variantes*** a los que aparecen después. Los campos variantes existirán según 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 sólo existe (y, por tanto, sólo es accesible) cuando el valor del discriminante (que, por lo demás, es un campo como cualquier otro) coincide con el valor de la correspondiente expresión constante:

> Deseamos construir un tipo que permita centralizar la información 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 esté 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;
    si estado: estado_civil es
      □ casado ⇒ pareja: nombre_completo;
      □ soltero ⇒ ; -- no hay nombre si no está casado
    fin si;
  fin tupla;

var p1,p2: persona;
```

p1

EMMA SÁNCHEZ RIUS		
21	*Enero*	*1950*
Soltero		

p2

ANA GÓMEZ GÓMEZ		
7	*Marzo*	*1957*
Casado		
PEDRO LÓPEZ CUETO		

Figura 68. ***Declaraciones y gráficos para un tipo* tupla *con variantes:*** Los gráficos 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

Así como el concepto de producto cartesiano tiene su correspondencia en el lenguaje UBL en el concepto de **tupla**, el concepto de unión 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 unión disjunta de los reales y los enteros), definiremos

```
tipo entero_o_real es
  tupla si es_real: logico es
    □ cierto ⇒ r: real;
    □ falso ⇒ e: entero;
  fin si; 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;
-- después de la ejecución de esta instrucción no tiene porque verificarse que x.r = 1.0
```

Instrucción con

La instrucción **con** sirve para poder referirse directamente a los campos de un objeto dado de tipo tupla, sin necesidad de indicar directamente en cada instrucción el nombre del objeto.

```
instrucción_con =
  con variable_tupla {,variable_tupla} haz
    instrucción
    {instrucción}
  fin [con];

variable_tupla = variable

Estilo: además del sugerido en la sintaxis,

con var {,var} haz instr {instr} fin;
```

***Figura 69. Sintaxis y estilo de la instrucción* con**

La instrucción **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 ***ámbito***), aplicando para la identificación de los objetos las siguientes convenciones:

- En el ámbito de la instrucción **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 rápida y cómoda de dar valor a f. Equivale a
f.d ← 28; f.m ← Febrero; f.a ← 1999;
```

- Si se produce alguna ambigüedad entre un campo y otra entidad en el ámbito de la instrucción **con**, se resuelve ésta 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 ámbito
fin con;
-- Aquí d es el caracter y no f.d
```

En este sentido, la instrucción **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 más instrucciones, si los tipos de O_i y O_j coinciden, la ambigüedad resultante de intentar decidir si un campo c es de O_i o de O_j se resuelve en favor del O más cercano textualmente entre los que preceden a la instrucción en cuestión; de otro modo, se toma el O_n con mayor n.

```
var f1,f2: fecha;
  ...
con f1 haz
  instrucción1;
  con f2 haz
    -- A es F2.A y no F1.A
  fin con;
  -- A es F1.A
fin con;
```

En los siguientes capítulos se encontrarán más ejemplos de programación 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 también a veces *periféricos*), como cintas magnéticas, discos, fichas perforadas, papel impreso u otros medios. Las posibilidades ofrecidas por los distintos periféricos varian según la naturaleza de éstos (por ejemplo, normalmente puede modificarse desde programa un fichero en disco magnético, pero no uno en fichas perforadas); aquí nos limitaremos a una presentación de lo que llamaremos *filas secuenciales*.

Un fichero puede concebirse como una secuencia de objetos, de longitud variable.

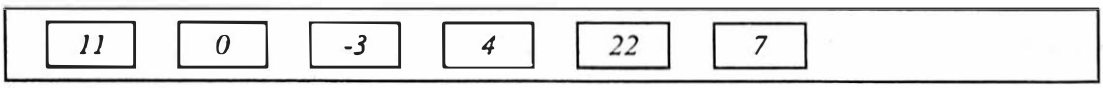


Figura 70. Un fichero de enteros

Estos objetos se encuentran físicamente "fuera" del alcance del intérprete del programa, excepto mediante las **filas**, métodos de acceso a los ficheros, y sus operaciones, que se definirán; su localización o *ubicación* podrá variar entre dos ejecuciones de un programa, sin afectar a sus resultados si los datos son los mismos, y podrá ser definida exteriormente al programa o mediante la acción predefinida *conecta*.

Acceso a los componentes de un fichero; estructura de fila

Distinguiremos los *ficheros*, grupos de datos contenidos en algún medio externo de almacenamiento, de las **filas**, estructuras del lenguaje que permitirán 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** o **conjunto** sin limitación alguna en el orden

de acceso (por ejemplo, después de acceder a *T[1]* puede accederse a *T[i]*, para cualquier *i* válido). En cambio, las **filas** son objetos que proporcionan un ***acceso secuencial*** y controlado a los elementos de un fichero.

El fichero de la Figura 70 en la página 167 puede manipularse mediante una variable de tipo **fila**:

var *f:* **fila de** *entero;*

Convendremos en que, en cada momento de la ejecución del programa, sólo es accesible a traves de la **fila** un único elemento del fichero, que llamaremos ***ventana*** o "***buffer***" de la **fila**; de otro modo, es como si "mirásemos" el fichero mediante una ventana, que forma parte de la fila. En la ventana sólo cabe un elemento del fichero.

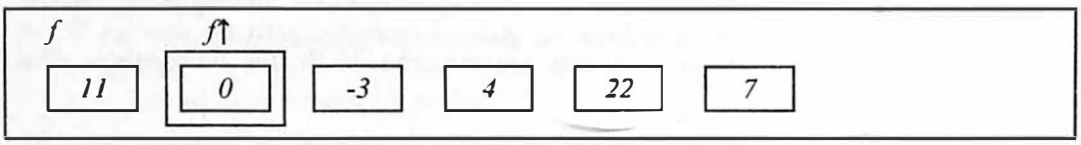


Figura 71. ***El mismo fichero, su* fila *y la ventana:*** (que en este caso está situada "sobre" el segundo elemento).

Para referirnos a la ventana, utilizaremos la notación *f*↑(si *f* es la **fila** en cuestión). *F*↑ es un modo de ***selección***, y por tanto puede utilizarse como cualquier objeto de su tipo: *T* si *f* es **fila de** *T*.

tipo_fila = **fila de** *tipo*

Figura 72. ***Sintaxis de un tipo* fila**

El efecto combinado de la inspección o modificación de la ventana y las acciones predefinidas que se explicarán, que "corren a la derecha" la ventana, permitirán la inspección (o ***lectura***) o la creación (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 ejecución del programa (o bien, si su ubicación 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 ejecución del programa.

Operaciones

Las siguientes operaciones predefinidas se aplican a los objetos de tipo **fila**:

- La asignación y la comparación *no son posibles* en el caso de las **filas**; consecuentemente, sólo es posible un parámetro de tipo **fila** si es un **var**-parámetro.

- La selección de la ventana, que, como hemos visto, se expresa *f*↑, sólo es válida si la **fila** está *conecta*da a un fichero.

- La **accion** predefinida *conecta* toma tres argumentos

  Conecta f,ubicación,modo;

  (donde *modo*={*lectura*|*escritura*}) y conecta lógicamente una **fila** con un fichero. *F* es la fila a conectar. *Ubicación* es una (expresión de tipo) tira que establece cuál es la ubicación física del fichero; esta tira es dependiente de Sistema Operativo y de Versión y no está definida por las reglas del lenguaje UBL. *Lectura* o *escritura* indican el modo de tratamiento del fichero. *Conecta* es la primera instrucción que debe ejecutarse con un fichero; funciona al modo de la ***inicialización*** que debe hacerse con los demás tipos de variable. Después de ejecutarse *conecta*, *f*↑, 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 está creando.

- Paralelamente, la **accion**

  Desconecta f;

  termina con la asociación entre la **fila** y el fichero. *F*↑ pasa a estar indefinido, y *f* queda lista para otra asociación, 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 ésta existe; si no existe, *f*↑ 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*↑ pasa a formar parte del fichero que se está creando, y la ventana avanza conceptualmente un espacio, quedando su valor indefinido, de modo que la siguiente operación *pon* coloque un valor a continuación de ése. Es incorrecto ejecutar *pon* si el modo es *lectura*.

- La **condicion**

  fdf(f)

  indica si la ventana está más allá del último elemento válido de la fila. No tiene sentido utilizarla en modo de *escritura*, ya que esto siempre sucede; en modo de *lectura*, indica si la última operación *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 último caso, *f↑* está 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 abreviación útil para las instrucciones

  v ← *f↑; obten f;*

  y suele utilizarse en vez de éstas en la mayoría de los casos.

- Igualmente,

  Escribe f,e;

  donde *e* ha de ser una expresión de tipo *T*, es abreviación de

  f↑ ← *e; pon f;*

El siguiente programa muestra la utilización de algunas de las operaciones descritas: calcula la media de los números 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;
  -- "?????" deberá substituirse por una tira adecuada, dependiente de versión
  mientras ~ fdf(f) haz
    suma ← suma + f↑;
    n ← n + 1;
    obten f;
  fin mientras;
  desconecta f;
  escribe "La media de los números 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 último elemento de un fichero, puede utilizarse la **condicion** predefinida *fdf*, o bien establecer algún tipo de convención privada: por ejemplo, puede convenirse en que el último elemento tendrá un valor válido para su tipo, pero determinado y absurdo para las condiciones del problema: en el Algoritmo 32, podría acordarse que la fila estaria terminada por el número 9999, y substituir

~ fdf(f)

por

f↑ ≠ 9999

A estas convenciones las llamaremos ***convenciones de fin de fila***; en un programa, especificaremos siempre cuál utilizamos, a menos que nos basemos en la operación predefinida *fdf*.

Fusión de ficheros

Un problema clásico al estudiar filas es el de ***fundir*** dos ficheros. Se suponen los dos ordenados (es decir: cada elemento es mayor o igual que el anterior, con arreglo a determinado criterio de ordenación), 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 fusión_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 contendrá el resultado de la fusión

 haz

  conecta f,'????',lectura;
  conecta g,'????',lectura;
  conecta h,'????',escritura;

  mientras ~fdf(f) ∧ ~fdf(g) haz
    si f↑ < g↑ entonces h↑ ← f↑; pon h; obten f; sino h↑ ← g↑; pon h; obten g; fin;
  fin mientras;

  si ~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. Fusión de ficheros

Filas de caracteres. El tipo predefinido *texto*

El caso de los ficheros de *caracteres*, que normalmente se utilizan para almacenar textos (en sentido amplio: textos escritos, programas, datos en forma legible, etc.), es más complejo y requiere de un tratamiento especial. Habitualmente se considera un texto como compuesto de una colección de líneas, compuestas a su vez de

caracteres; e interesa poder tener en cuenta la estructura de líneas, y no sólo 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 secuencia de lineas; cada línea (al ser examinada o creada; y automáticamente, como se verá) contendrá un carácter adicional a la derecha, llamado ***carácter de fin de línea***, que "funcionará" como si fuese un espacio en blanco al posicionarse la ventana encima.

```
ésta es la primera línea del fichero$
de caracteres$
     $
la línea anterior contiene exactamente$
cinco espacios$
```

Figura 73. ***Un fichero de texto:*** El carácter de fin de línea se representa mediante el carácter "$".

Este carácter tiene el efecto adicional de que, al ser examinado mediante la ventana, la condición predefinida *fdl(f)* pasa a valer *cierto* (y sólo lo hace en esos casos).

Las operaciones aplicables a los objetos de tipo *texto* (además de las que se aplican a todos los de tipo **fila**) son

- La condición *fdl* (Fin De Línea)

  condicion *fdl*(**var** *f: Texto);*

  que vale *cierto* si y sólo si la ventana está posicionada sobre el carácter de fin de línea. En ese caso, y por convención, 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 después del primer carácter de fin de línea que encuentre, lo cual equivale lógicamente a pasar a principio de la línea siguiente (si existe, y no hay fin de fila).

- En modo de *escritura*, la

  accion *escribe_linea*(**var** *f: Texto);*

asigna un carácter de fin de línea a la ventana y lo *pone* en el fichero, cambiando efectivamente de linea y empezando una nueva. Este es el único modo de escribir un carácter de fin de línea desde un programa.

Además, para filas de tipo *texto* son válidas las siguientes convenciones

- A diferencia de los demás tipos fila, para los cuales es obligatoria la coincidencia de tipos entre el objeto o expresión 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 secuencia 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 notará que esta definición respeta y coincide con los métodos de lectura y escritura que se han utilizado informalmente en los capítulos anteriores.

- Una instrucción como

  escribe $f,v_1,v_2,\ldots,v_n;$

  donde f es un *texto* y v_i expresiones, se considera abreviación de

  escribe f,v_1; *escribe* f,v_2; ... *escribe* f,v_n;

  y similarmente para la instrucción *lee*.

- Se admiten parámetros adicionales en las instrucciones *lee_linea* y *escribe_linea*:

  lee_linea $f,o_1,o_2,\ldots,o_n;$

  se considera equivalente a

  lee $f,o_1,o_2,\ldots,o_n$; *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 último, se consideran predefinidos los objetos

var *entrada, salida: Texto;*

y se considera que, antes de la ejecución de la primera instrucción del programa, se ejecuta

conecta entrada,'?????',lectura;
conecta salida,'?????',escritura;

prohibiendo la ejecución de cualquier instrucción *conecta* o *desconecta* sobre esos objetos. Además, 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 ~ fdf haz
    mientras ~ 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:* Nótese el uso de abreviaciones: *fdl* significa *fdl(entrada)*, etc...

Control de Stocks

El siguiente programa actualiza un fichero de stocks. Para cada artículo, el fichero (que llamaremos ***fichero maestro***) contiene un número que lo identifica, un descripción de como máximo 40 caracteres, y un número que indica su cantidad (se supone el fichero maestro ordenado respecto del número de identificación). Cada día, se lleva control de los artículos que se compran y venden, de modo que se genera un ***fichero de actualizaciones***. Se distinguen entre tres tipos de actualización, identificados mediante una clave alfabética: añadir (A) un artículo al stock, suprimirlo (S) o cambiarlo (C) -- ya sea en la cantidad o en su descripción. Cada día se ordena por número de artículo 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 erróneas. La convención de fin de fila está explicada en el programa.

```
programa actualización es
  tipo nombre_artículo es :tira(40);
  tipo ficha es tupla número: entero; descripción: nombre_artículo; cantidad: entero;
fin;
  tipo operación es {A,S,C}; -- Añadir, Cambiar, Suprimir
  var entrada,salida,entrada2,salida2: texto; fm,fa: ficha; op: operación;
    (* convención de fin de fila: la última ficha de los ficheros maestro y de
actualización *)
  const fin_de_fila = 999999; (* indican el fin de fila al tener como número 999999
*)

  accion lee_maestro (var f: ficha) haz
    lee_linea entrada,f.número:10,f.descripción:40,f.cantidad:10;
  fin lee_maestro;

  accion lee_actualización (var op: operación; var f: ficha) haz
    lee entrada2,op:1,f.número:10;
    si op en {C,A} entonces lee entrada2,f.descripción:40,f.cantidad:10; fin;
    lee_linea entrada2;
  fin lee_actualización;

  accion pon_maestro (const f: ficha) haz
    escribe_linea salida,f.número:10,f.descripción,f.cantidad:10;
  fin pon_maestro;

  accion pon_error (const op:operación; const f: ficha) haz
    escribe_linea salida2,op:1,f.número:10,f.descripción,f.cantidad:10;
  fin pon_error;

 haz
  conecta entrada,'????',lectura; conecta entrada2,'????',lectura;
  conecta salida,'????',escritura; conecta salida2,'????',escritura;
  lee_maestro fm; lee_actualización op,fa;
  repite
    mientras fm.número < fa.número haz pon_maestro fm; lee_maestro fm; fin;
    si fm.número = fa.número entonces -- Cambiar, Suprimir
      si op es
        □ A ⇒ pon_error op,fa; lee_actualización op,fa;
        □ C ⇒
          repite
            fm ← fa; lee_actualización op,fa;
          hastaque op ≠ C ∨ fm.número ≠ fa.número;
        □ S ⇒ lee_maestro fm; lee_actualización op,fa;
      fin si;
    sino -- Añadir
      repite
        si op = A entonces pon_maestro fa; sino pon_error op,fa; fin;
        lee_actualización op,fa;
```

```
    hastaque fa.número ≥ fm.número;
  fin si;
hastaque fm.número = fin_de_fila ∨ fa.número = fin_de_fila;
si
  □ fa.número ≠ fin_de_fila ⇒
    repite
      si op = A entonces pon_maestro fa; sino pon_error op,fa; fin;
      lee_actualización op,fa;
    hastaque fa.número = fin_de_fila;
  □ fm.número ≠ fin_de_fila ⇒
    repite
      pon_maestro fin; lee_maestro fm;
    hastaque fm.número = fin_de_fila;
fin si;
pon_maestro fin;
desconecta entrada; desconecta entrada2; desconecta salida; desconecta salida2;
fin programa;
```

Algoritmo 35. Control de Stocks

Subprogramas como parámetros

Además de objetos (pasados por copia, por variable y por constante), un subprograma puede admitir entre sus parámetros lo que llamaremos ***subprogramas formales***; este tipo de parámetro se utiliza para diseñar subprogramas de algún modo ***genéricos***, en el sentido de que, dependiendo del ***subprograma actual*** pasado como argumento en la invocación, actuarán (o realizarán evaluaciones; o tratarán sucesiones) de formas distintas.

parámetros_por_subprograma = *cabecera*

Figura 74. Sintaxis de un parámetro por subprograma

Son ejemplos de subprogramas con parámetros por subprograma

accion *A* (**accion** *B*; **funcion** *H*: *real*);
funcion *F* (*x*: *real*; **funcion** *G*(*z*: *real*): *real*): *entero*;
sucesion *S* (**sucesion** *T*: *real*): *real*;

Una lista de parámetros puede contener parámetros por subprograma de cualquier tipo, arbitrariamente mezclados con parámetros por copia o por nombre. Las cabeceras que declaran los parámetros por subprograma pueden contener a su vez parámetros; los nombres de estos "parámetros de segundo orden" (p. ej., el parámetro *z* de la función parámetrica *G* de la función *F*) pueden ser arbitrarios, y no sirven más que para indicar el tipo de los parámetros del subprograma paramétrico.

Los subprogramas actuales pasados como argumento deben tener listas de paramétros ***compatibles*** con las de los parametros declarados, en el sentido de que los tipos de sus parámetros deben ser idénticos y estar en el mismo orden (o, si se da el caso, los parámetros por subprograma deben ser compatibles), aunque los identificadores de los parámetros difieran.

Por ejemplo, una

funcion *F* (*x*: *real*; **funcion** *G*(*z*: *real*): *real*): *entero*;

puede ser invocada como

F(3.14,h)

y producirá un valor entero, siempre que la declaración del argumento o subprograma actual *h* tenga una lista de parámetros compatible con la de *G*; por ejemplo, si esta declarada como

funcion *h(r: real): real*

(Nótese que el parámetro de *h* se llama *r* y el de *G* se llama *x*; esto no tiene importancia, ya que los dos son de tipo *real*).

Integración por trapecios

Una de las formas mas simples de integrar númericamente una

funcion *f(x: real): real;*

entre los valores *a* y *b* es el ***método 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 área 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 evalúa como la suma de las integrales aproximadas de los intervalos, es decir, como

$$I = \Delta \times ((f(x_0)+f(x_n))/2 + f(x_1) + f(x_2) + \ldots + f(x_{n-1}))$$

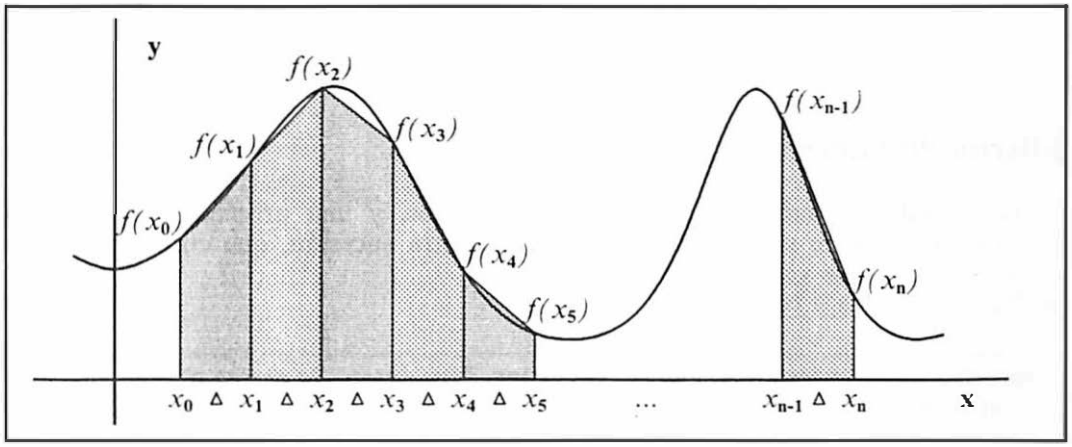


Figura 75. Ejemplo de integracion por trapecios.

La siguiente función implementa ese método:

```
funcion integral(funcion f(x:real): real; a,b: real; n: entero): real es
  var i,i2,il,delta,Xi: real; k: entero;
haz
  il ← 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*(il/2 + i2);
  vale i;
fin integral;
```

Algoritmo 36. Integración numérica por el método de los trapecios

y se utiliza, dada una función 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 sucesiones

Dada cualquier sucesión S de números enteros, y una propiedad P sobre los enteros, el siguiente subprograma implementa otra sucesión, que contiene solo los elementos de S que verifican P

```
sucesion filtro(sucesion 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 sucesiones de enteros

El método es general, y da una idea de cómo escribir filtros abstractos arbitrarios que restrinjan sucesiones según cualquier criterio.

Recursividad

Introducción

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 invocación); lo que no hamos visto es qué pasa si un subprograma se invoca (directa o indirectamente) a sí mismo. Cuando sucede esto último, se dice que el subprograma es ***recursivo***, y que la invocación que de sí mismo hace es una ***invocación recursiva***. El lenguaje trata este tipo de invocaciones sin inconvenientes, aunque se plantean algunos problemas, que intentaremos aclarar en esta discusión, 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 representación en forma de caracteres:

En efecto, una invocación 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 será nulo).
   b. Y a continuación 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 también para cualquier otro argumento, como puede deducirse del examen del subprograma).

Éste no es por sí mismo un buen ejemplo de uso de recursividad, ya que el mismo efecto podría haber sido obtenido (además, de un modo más claro) mediante una iteración (cuya escritura se plantea como ejercicio); sin embargo, muestra el mecanismo de la recursividad, y nos permite plantear algunas cuestiones.

- Cada invocación (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 invocación [recursiva] es *distinta* de la *i* de la primera invocación); 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 confusión entre las *i*s, ya que en la segunda instancia sólo podemos manipular la allí 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 invocación a una acción 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 metodología de diseño descendente: cada abstracción se refina en función del vocabulario existente, y por tanto es posible utilizar alguna abstracción 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 "más 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 "más pequeñas" o "que den menos trabajo"; y que "tiendan" a
- la (o las) subtarea(s) "mínimas", que serán resueltas sin la ayuda de la recursión.

Llamaremos ***grafo*** o ***árbol de invocaciones*** a la representación arbórea de las instancias producidas por una serie de invocaciones. Por ejemplo, el grafo de invocaciones de

recursiva 154;

será, representando en cada nodo el correspondiente argumento,

recursiva 154 ——> *recursiva* 15 ——> *recursiva* 1 ——> *recursiva* 0

y el efecto se seguirá del recorrido inverso del grafo

escribe 4 <—— *escribe 5* <—— *escribe 1* <—— **nada**;

Para reproducir de otro modo las condiciones del párrafo anterior: ha de garantizarse la finitud del grafo para cualesquiera argumentos del subprograma.

Recursión directa e indirecta. Definiciones e implementaciones de subprogramas

Un subprograma es ***directamente recursivo*** cuando se invoca a sí 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; -- recursión indirecta
  ...
fin A;
```

Si los subprogramas están declarados en el mismo nivel declarativo o son ***mutuamente recursivos*** (ésto 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 declaración de *A* y *B* que satisfaga la regla de "declaración antes de uso", a menos que utilizemos definiciones e implementaciones:

```
accion A; -- definición de A

accion B es ... haz ... A; ... fin; -- implementación de B.
-- Usa A, puesto que ya está definido, pero debe darse su implementación

accion A es ... haz ... B; ... fin;
-- implementación de A. En este caso, B está también
-- implementado, además de definido.
```

La sucesión de Fibonacci

La sucesión 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 sucesión *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 v i = 1 entonces vale 1;
  sino vale Fibonacci(i-1) + Fibonacci(i-2);
  fin si;
fin Fibonacci;
```

Algoritmo 38. La sucesión de Fibonacci

Desde el punto de vista de la eficiencia, el método es desastroso, como puede verse en el siguiente árbol de invocaciones, donde en cada punto se ha escrito sólo el valor del parámetro:

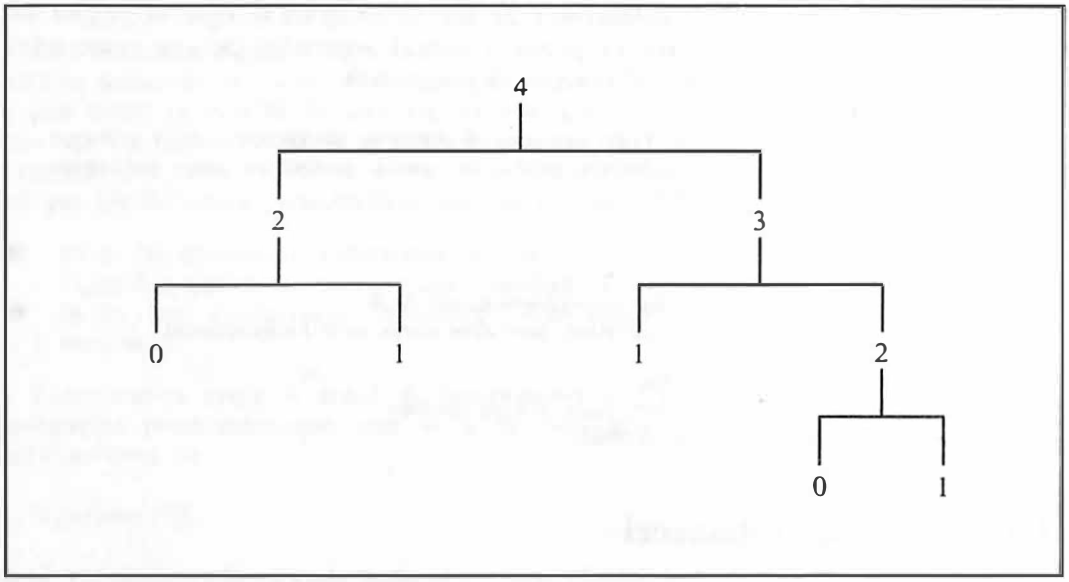


Figura 76. Arbol de invocaciones para Fibonacci(4): Cada camino se detiene al llegar a I=0 o I=1.

y debe ser substituido por un esquema iterativo:

```
funcion Fibonacci(i: entero) es
  var f,fant: entero;
haz
  si i = 0 ∨ 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 sucesión de Fibonacci, versión iterativa

Las torres de Hanoi

Se dispone de tres estacas, llamadas A, B y C, y de una colección 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 esté encima. Se trata de trasladar los discos de la estaca A a la estaca B, usando la C como auxiliar si es preciso, dejándolos en el mismo orden en que se encuentran, y utilizando cualquier número de veces las siguientes reglas de movimiento:
>
> - En cada movimiento se puede cambiar sólo 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 situación en la que un disco grande descanse sobre uno más pequeño.

y tiene solución, como se demuestra fácilmente por inducción:

- Para $n = 0$ trivialmente hay una solución, ya que sin hacer nada queda resuelto el problema.
- Suponiendo 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 hipótesis de inducción).
  2. Seguidamente, se mueve el último disco de la estaca A a la B (lo cual es posible, ya que no contraviene ninguna de las reglas prescritas).
  3. Por último, se trasladan los $n-1$ discos restantes de la estaca C a la B, con lo cual queda ultimada la prueba de existencia de solución para n discos.

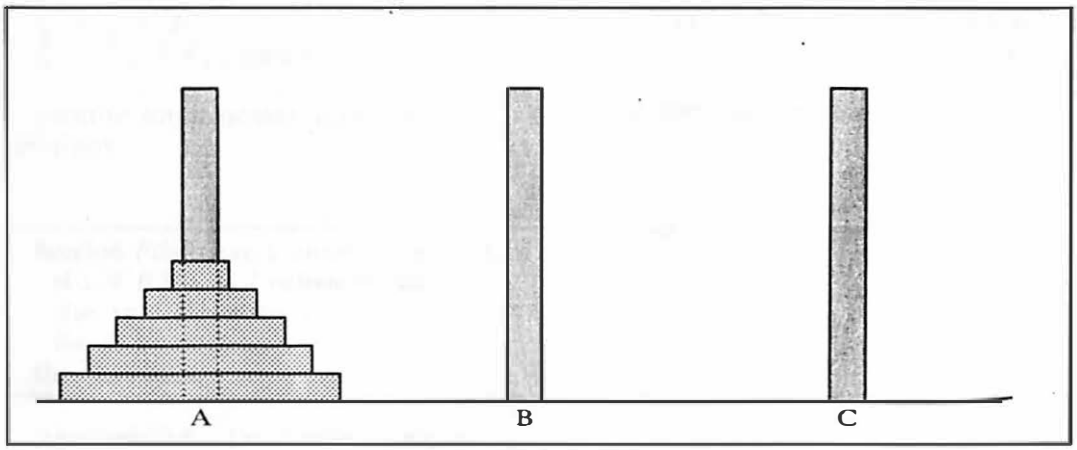


Figura 77. Las Torres de Hanoi: posición inicial con 5 discos

De la demostración 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 gráficamente los movimientos necesarios para $n = 2$.

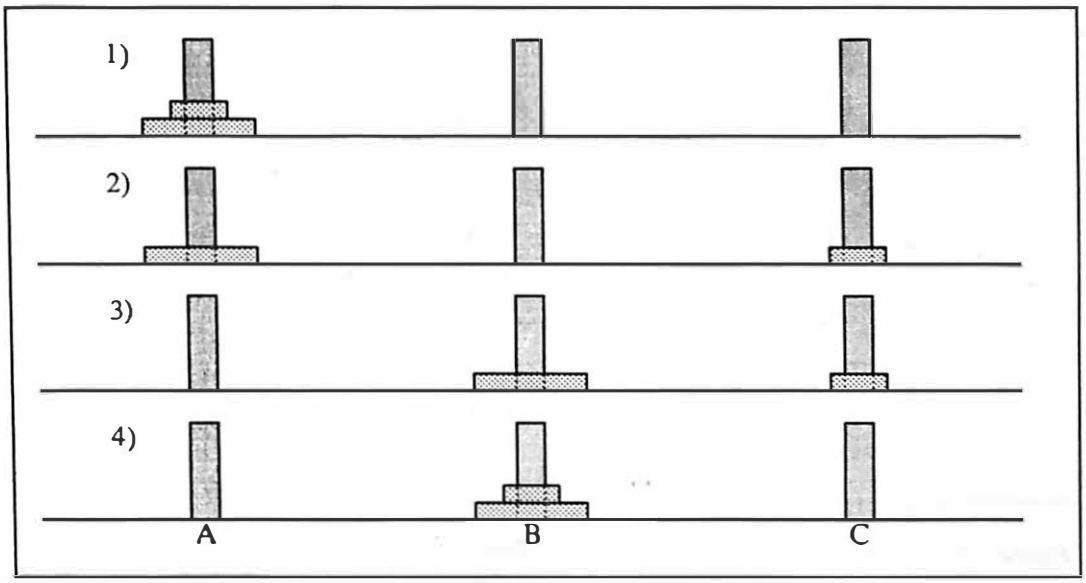


Figura 78. ***Movimientos para el caso de dos discos:***

1. Posición inicial
2. Primer movimiento: de *A* a *C*
3. Segundo movimiento: de *A* a *B*
4. Tercer y último movimiento: de *C* a *B*

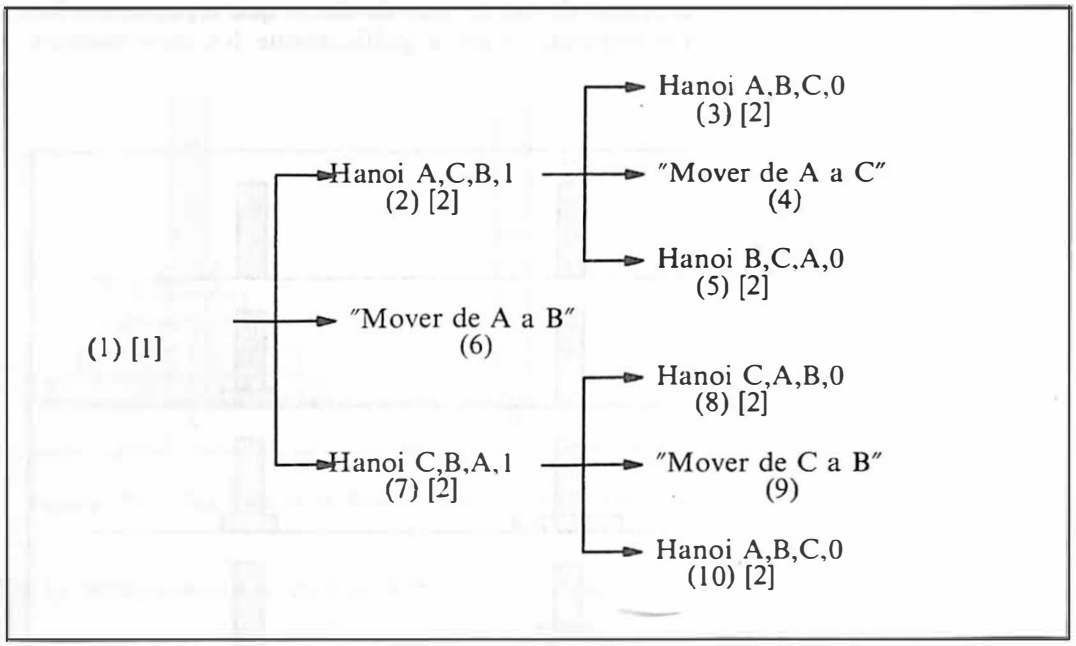


Figura 79. ***Grafo de invocaciones para Hanoi A,B,C,2:*** Los números entre paréntesis indican el orden en el que se ejecutan las instrucciones; los que están entre corchetes, el número de instancias activas de la **accion** *Hanoi*. Las invocaciones cuyo cuarto argumento es 0 no se han expandido, ya que su efecto es nulo.

Descomposición 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 última 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 "número de dados con valor i".

Una aproximación sistemática al problema permite ver que n_6 podrá tomar cualquier valor entre 0 y p **div** 6, quedando condicionados los demás n_i por la elección que se haga a descomponer el valor $p - 6*n_6$; el mismo razonamiento, aplicado a ese valor, permite obtener, para cada elección de n_6, un conjunto de elecciones posibles para n_5, y un nuevo resto

$$p - 6*n_6 - 5*n_5$$

y así, hasta llegar a

$$p - 6*n_6 - 5*n_5 - 4*n_4 - 3*n_3 - 2*n_2$$

que está determinado y se obtiene directamente, sin elección.

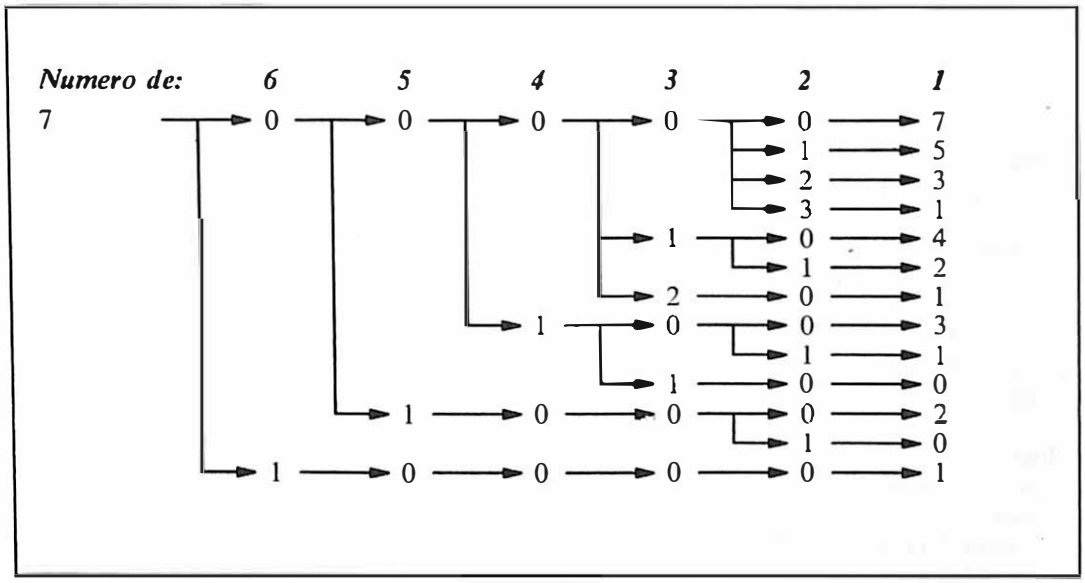


Figura 80. Descomposiciones posibles del número 7

Del análisis 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 descompón(p,i: entero) es

    var k: entero;

    accion solución 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 solución;

  haz
    si i = 1 entonces N[1] ← p; solución;
    sino
      para k en asc(0,p div i) haz
        N[i] ← k;
        si k*i = p entonces solución; sino descompón p - k*i,i-1; fin;
      fin para;
    fin si;
  fin descompón;

haz
  lee Pp; descompón Pp,6;
fin programa;
```

Algoritmo 41. Descomposición en suma de dados

Las Ocho Reinas, en versión recursiva

El problema de las Ocho Reinas (véase el Algoritmo 27 en la página 150) admite también una formulación recursiva. Basta con considerarlo como consistente en "intentar" poner las ocho reinas, dando en este caso a "intentar" un significado recursivo: "intentar poner una reina" (en una fila dada) será "poner una reina en cada posición no conflictiva" (de su fila) seguido cada vez de "intentar poner una reina" (en la siguiente fila). De este razonamiento se deriva el siguiente programa, que resulta ser mucho más simple y corto que su correspondiente versión iterativa.

```
programa Ocho_damas es

 var da: tabla [-7..7] de logico;
 var dd: tabla [2..16] de logico;
 var col: tabla [1..8] de 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 solución es var i: entero; haz
   para i en asc(1,8) haz escribe ' ',D[i]; fin; escribe_linea;
 fin solución;

 accion intenta(i: entero) es
   var j: entero;
 haz
   para j en asc(1,8) talque ~(col[j] ∨ da[i-j] ∨ dd[i+j]) haz
     D[i] ← j;
     col[j] ← cierto; da[i-j] ← cierto; dd[i+j] ← cierto;
     si i = 8 entonces solución; sino intenta i+1; fin;
     col[j] ← falso; da[i-j] ← falso; dd[i+j] ← falso;
   fin para;
 fin intenta;

haz
  inicializa; intenta 1;
fin programa;
```

Algoritmo 42. Las Ocho Reinas en versión recursiva

Ejercicios

1. Reprogramar la **accion** *recursiva* de la introducción en forma iterativa.

2. Justificar en el Algoritmo 41 en la página 192 el uso de la variable lógica *primer_ni* y la posición de la **accion** *solución*.

3. La ***función 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 81. ***Un objeto de tipo*** **nombre** ***y el objeto nombrado:*** En este caso, el tipo de *n* es **nombre** *entero*. *ß* 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, *ß*) puede tratarse mediante la notación *n*↑. La flecha dibujada en la figura indica la relación conceptual que existe entre los dos objetos: por esta razón 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 declaración del tipo

tipo *identificador* **es nombre** *tipo;*

Figura 82. ***Sintaxis de una declaración de tipo*** **nombre**

Por ejemplo, el objeto *n* de la Figura 81 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, además del valor **nulo** (**nulo** es un identificador reservado), que puede entenderse como "el nombre de ningún objeto"; si *n* vale **nulo**, es un error hablar de *n*↑. En cuanto a los valores que no son **nulo**, la única forma de dar valor a un objeto de tipo **nombre**, aparte de la asignación, es utilizando la **accion** predefinida *crea*:

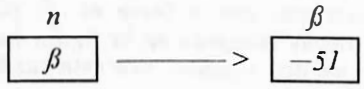
crea n;

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*↑, está indefinido hasta que se le de valor:

n↑ ← *-51*



y, a diferencia de los objetos declarados como **var**iables, no ve limitada su existencia por ningun ámbito sintáctico. Puede ser destruido mediante la instrucción

libera n;

cuyo efecto es destruir la variable *ß*, con lo que *n*↑ deja de ser válido.

Dada una declaración como

var *i: entero;*

podría pensarse en asignar

n ← *i; -- incorrecto*

con la intención de que *n* "apuntase" a *i*; sin embargo, tal asignación 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 sí puede hacerse sin problema es

n↑ ← *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.

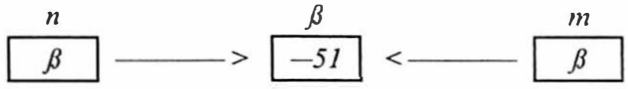
Además, si suponemos declarado

var *m: T;*

es posible asignar

m ← *n;*

con lo que se llega a la situación representada en la siguiente figura:



Dos objetos de tipo **nombre** pueden "apuntar" pues al mismo objeto.

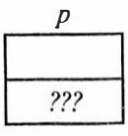
Principal uso de los objetos de tipo nombre

El interés 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*:

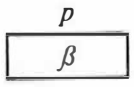


campo a, *de tipo* entero
campo b, *de tipo* erroneo: *cuál es su forma?*

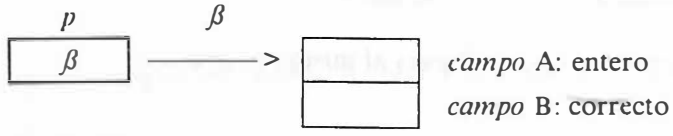
Diremos que un tal tipo es ***directamente recursivo***, y prohibiremos su uso y declaración; sin embargo, si declaramos

```
tipo correcto es nombre
  tupla
    a: entero;
    b: correcto;
  fin tupla;
```

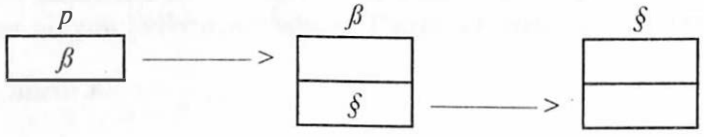
utilizando en este caso ***recursión indirecta***, no hay ningun problema al considerar la forma de un objeto *p* de tipo *correcto*:



ß será, 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 podrá 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 también ser **nulo**, o bien el nombre de otra tupla (dado que su tipo es *correcto*):



Lo cual permitirá 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; además, y puesto que al valor de *p* (o del campo *b* de cualquier tupla) puede ser **nulo**, se ve también que el número de componentes de esa lista es manipulable (ya que la presencia del valor **nulo** indica, por convención, 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 declaración de un único objeto de tipo **nombre** permite el acceso y la manipulación de una lista arbitrariamente larga de (tuplas que contienen, en este caso) enteros. En general, y considerando la posibilidad de declarar más campos, es posible manipular grafos arbitrariamente complejos. Estudiaremos aquí algunos de los más utilizados en programación.

Listas unidireccionales

La forma más 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 únicamente como ejemplo; la teoría expuesta se generaliza sin dificultad para otros tipos. En el resto de este párrafo, se considera declarado

```
var p: lista;
```

La representación de (un posible valor de) *p* será

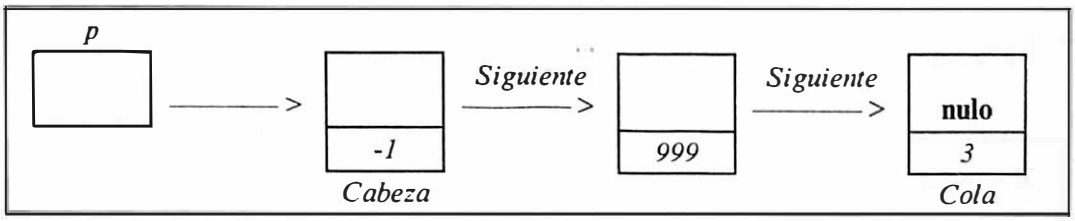


Figura 83. Una lista unidireccional de enteros

Llamaremos ***cabeza*** al principio de la lista (esto es, al valor de *p*) y ***cola*** a su final. Diremos que la lista está ***ordenada*** si lo están sus 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; -- encadénalo en la lista
p ← q; -- modifica el valor de la cabeza
```

Obsérvese que el método vale para todos los casos, incluido aquél en que la lista es vacía, o, lo que es lo mismo, *p* = **nulo**.

Para ***insertar*** un elemento de nombre *q* ***después*** de una posición arbitraria de nombre *r*, bastará con seguir el método indicado en la siguiente figura:

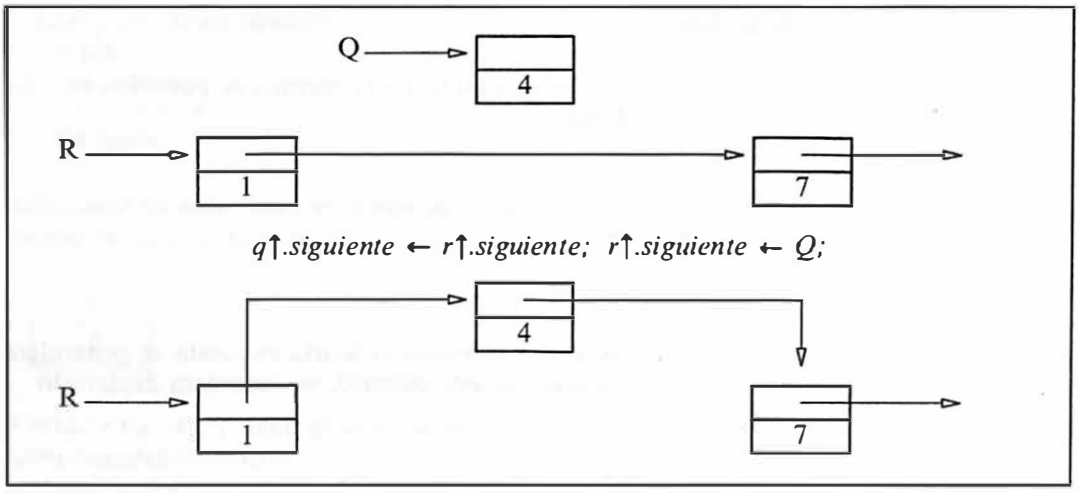


Figura 84. Inserción en una lista unidireccional

La ***supresión*** del *siguiente* de un elemento de nombre *q* es igualmente sencilla.

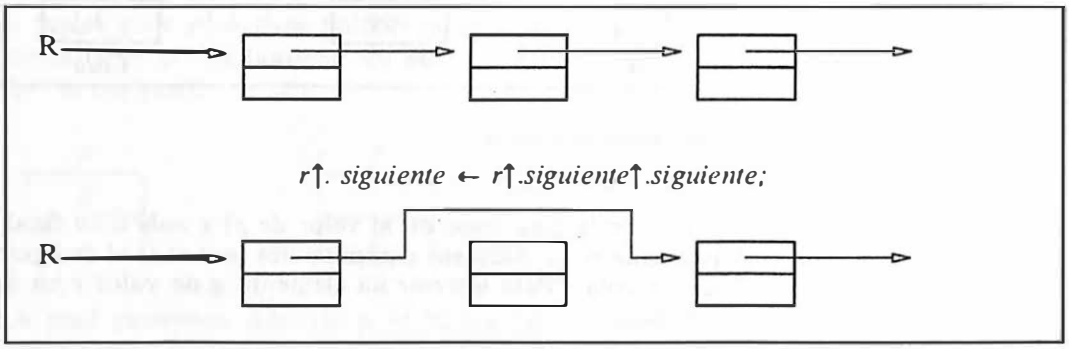


Figura 85. Supresión del siguiente de un elemento en una lista unidireccional

pero no lo es tanto la supresión del propio *q*↑, que se plantea como ejercicio.

Es trivial diseñar **sucesiones** para tratar los elementos de una lista:

```
-- genera nombres de los elementos de la lista
sucesion 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
sucesion val(l: lista): entero haz
  mientras l ≠ nulo haz
    produce l↑.e;
    l ← l↑.siguiente;
  fin mientras;
fin val;
```

Algoritmo 43. Sucesiones para el tratamiento de listas unidireccionales

La búsqueda de (el nombre de) un elemento con un valor entero prefijado n puede expresarse

```
si existe q en elem(p) talque q↑.e = n entonces ...
```

o, sin utilizar las sucesiones, como

```
q ← p;
mientras q ≠ nulo ∧∧ q↑.e ≠ n haz -- Nótese la necesidad de ∧∧
  q ← q↑.siguiente;
fin mientras;
-- q = nulo ∨ q↑.e = n
```

Arboles

Un ***árbol binario*** (o simplemente ***árbol***) se define (recursivamente) como

Un árbol es
- *o el* ***árbol vacío*** *(que no consta de ningun elemento)*
- *o un elemento (llamado* ***nodo****)*
  - ***y*** *dos árboles (****subárbol izquierdo*** *y* ***subárbol derecho****)*

o, en UBL,

```
tipo árbol es nombre
  tupla
    nodo: entero;
    sub_i,sub_d: árbol;
  fin tupla;
```

De la definición se deduce (suponiendo que los elementos o nodos son enteros, que **nulo** simboliza el árbol vacío, y que *ARBOL(elemento,arboll,arboll)* es el árbol cuyo nodo es *elemento* y sus subárboles *arboll* y *arbol2*) que son árboles 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 árboles suelen representarse en forma de árbol invertido (de ahí su nombre). Al nodo situado "más arriba" se le suele llamar ***raíz***; y a los subárboles situados "más abajo" (esto es, aquellos cuyos dos subárboles son el árbol vacío), ***hojas***.

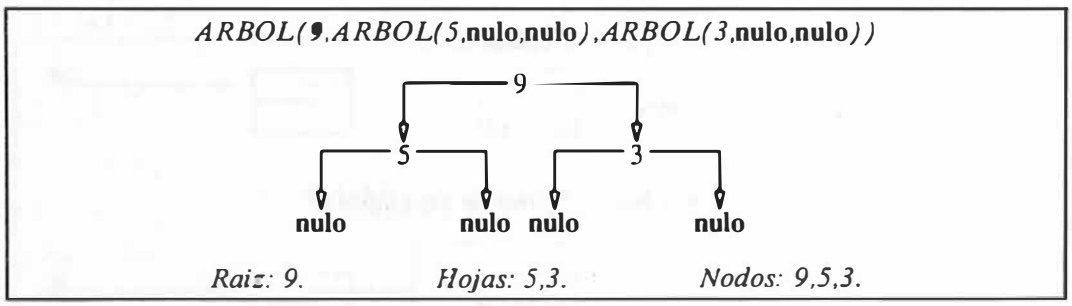


Figura 86. Representación de un árbol binario

Las operaciones más usuales sobre árboles incluyen funciones para hallar el valor de la raíz, y los subárboles de un árbol dado,

```
funcion raíz(a: árbol): entero haz vale a↑.nodo; fin;

funcion sub_i(a: árbol): árbol haz vale a↑.sub_i; fin;

funcion sub_d(a: árbol): árbol haz vale a↑.sub_d; fin;
```

Algoritmo 44. ***Funciones para el manejo de árboles:*** En todos los casos se ha supuesto a ≠ **nulo**; el tipo del campo *nodo (entero)* se utiliza únicamente como ejemplo.

acciones para insertar elementos en el árbol según determinado criterio

```
accion inserta(n: entero; var a: árbol) haz
  -- Inserta un nodo de valor N en el árbol A suponiendo que los valores
  -- de todo subárbol izquierdo son menores que el nodo del que dependen,
  -- y que éste es menor que los valores de los nodos de su subárbol derecho.
  -- Si ya existe un nodo con valor N, el efecto es nulo.
  si
    □ a = nulo ⇒
      crea a;
      con a↑ haz
        nodo ← n;
        sub_i ← nulo;
        sub_d ← nulo;
      fin con;
    □ a↑.nodo > n ⇒ inserta n, a↑.sub_i;
    □ a↑.nodo < n ⇒ inserta n, a↑.sub_d;
    □ otros ⇒ nada;
  fin si;
fin inserta;
```

Algoritmo 45. ***Inserción en un árbol binario***

y sucesiones para "recorrer" el árbol en algún orden. Los órdenes más conocidos para recorrer un árbol son los siguientes (su definición, como la del árbol, es también recursiva):

- Recorrer un arbol en ***inorden*** es

  (1) recorrer el subárbol izquierdo
  (2) pasar por la raiz
  (3) recorrer el subárbol derecho

- Recorrer un árbol en ***preorden*** es

(1) pasar por la raíz
(2) recorrer el subárbol izquierdo
(3) recorrer el subárbol derecho

- Recorrer un árbol en ***postorden*** es

(1) recorrer el subárbol izquierdo
(2) recorrer el subárbol derecho
(3) pasar por la raíz

La siguiente figura muestra un árbol y los valores obtenidos recorriéndolo en preorden, postorden e inorden:

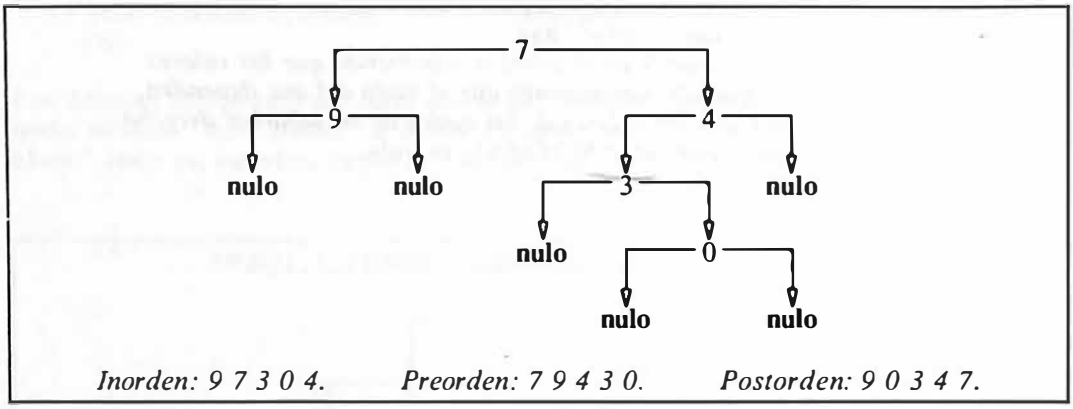


Figura 87. Un árbol y sus recorridos

Los recorridos se implementan mediante sucesiones, que presentamos aquí en su versión recursiva

```
sucesion inorden(a: árbol): 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;

sucesion preorden(a: árbol): 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;

sucesion postorden(a: árbol): 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. Sucesiones inorden, preorden y postreorden sobre un árbol

En el próximo capítulo, al hablar de **modulos**, se estudiarán ejemplos prácticos de utilización de **nombres**.

Ejercicios

1. Diseñar un algoritmo para insertar elementos en la cola de un lista unidireccional [**Indicación**: 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 [**Indicación**: utilizar, si es preciso, variables auxiliares].

3. Escribir una **accion** que tome como parámetro (por variable) la cabeza de una lista unidireccional, invierta esa lista, y modifique el parámetro para que siga apuntando a la cabeza de la lista invertida (lo que antes era la cola).

4. Escribir una **sucesion** no recursiva para atravesar un árbol binario en inorden (**Indicaciones**: puede probarse de ampliar la definición de tipo *árbol* para que incluya el **nombre** del nodo del cual cada nodo "desciende").

Módulos

Los **modulos** son mecanismos sintácticos generales que pueden utilizarse con muy diversos propósitos: encapsulación de entidades lógicamente relacionadas, protección de la estructura interna de un tipo de datos, control selectivo de la visibilidad de identificadores, etc. Sintácticamente, constan de dos partes, que llamaremos ***parte de definición*** y ***parte de implementación***.

```
declaración_de_módulo = definición_de_módulo | implementación_de_módulo

definición_de_módulo =
  modulo identificador es
    definición
    {definición}
  fin [identificador];

definición = declaración_de_objeto
  | declaración_de_tipo
  | cabecera ;
  | definición_de_módulo

implementación_de_módulo =
  modulo implementa identificador es
    {declaración}
  [haz
    instrucción
    {instrucción}]
  fin [identificador];
```

***Figura 88. Sintaxis de las declaraciones de* modulo**

La parte de definición puede contener declaraciones de objetos (constantes o variables), de tipos, ***definiciones*** de subprogramas (vease "Recursividad" en la página 183) y partes de definición de otros módulos; la parte de implementación (que puede omitirse si la parte de definición consta sólo de declaraciones de objetos) *debe* contener las implementaciones de los subprogramas, y las partes de

implementación de los módulos, definidos en la parte de definición, y *puede* contener declaraciones adicionales de objetos, tipos, módulos o subprogramas "internos" (el sentido en el que decimos "internos" quedara claro en seguida), así como una parte ejecutable, que puede utilizarse, por ejemplo, para inicializar los objetos declarados. Las partes ejecutables de los módulos se ejecutan, en el orden en que están declarados, inmediatamente después de la activación del bloque que los declara.

Las dos partes de un módulo deben hallarse en la misma parte declarativa, la definición antes de la implementación, pero no necesariamente juntas (esto es, pueden encontrarse otras declaraciones entre la definición y la implementación).

Las reglas de reconocimiento de nombres se alteran al utilizar **modulos**. Los identificadores declarados en la parte de definición pueden utilizarse si media una declaración **usa**: en este sentido, los identificadores pueden "salir" de los paréntesis sintácticos en los que se definen, cosa que no sucede más que con **modulos**; los declarados en la implementación son siempre locales al **modulo**, y no pueden utilizarse desde otra parte.

Las entidades declaradas en un módulo pueden hacerse ***visibles*** (o sea, conocidas o accesibles) en un bloque mediante la declaración **usa**:

declaración_usa = **usa** *identificador* {,*identificador*};

Figura 89. Sintaxis de la declaración **usa**

Esta declaración puede utilizarse inmediatamente después del módulo de definición (y antes de la implementación), o en cualquier otro sitio; su efecto es añadir al espacio de identificadores utilizables el de las entidades definidas en el módulo. Las únicas entidades visibles (a través de la declaración **usa**) son las de la parte de definición, mientras que las de la parte de implementación quedan ocultas, y no pueden utilizarse fuera del módulo. Este mecanismo permite, entre otras cosas, el ***control selectivo de la visibilidad*** que se había 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 estén usando 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 quién?)
fin A3;
```

Un caso típico en que los módulos son útiles es al diseñar un "paquete" de subprogramas relacionados con algún tema: por ejemplo, un conjunto de subprogramas matemáticos:

```
modulo Funciones_matemáticas_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 hiperbólico
  funcion cosenh( x: real ): real; -- coseno hiperbólico
  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; -- raíz cuadrada

fin Funciones_matemáticas_elementales;

modulo implementa Funciones_matemáticas_elementales es
  -- aquí se escribirían las implementaciones de
  -- cada uno de los subprogramas definidos más arriba
fin Funciones_matemáticas_elementales;
```

Algoritmo 47. Subprogramas matemáticos

Lo interesante es que un programador que ***utilice*** (puede ser una persona distinta del que lo escribe) el **modulo** anterior sólo debe "entender" (conocer el significado de) la parte de definición, y por tanto puede prescindir de la implementación; de hecho, si conviene, quien escribió el **modulo** puede cambiar (o corregir, si se revelase errónea) la implementación de cualquier subprograma (por ejemplo, substituyéndola por otra más eficiente o más exacta) sin que los que lo **usa**n observen cambio alguno. Todo lo cual es idóneo como soporte de metodologías modulares de programación.

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 aparición de cada una de las palabras del texto (no se ha tenido en cuenta la presencia de signos de puntuación, pero el programa puede readaptarse fácilmente).

El método utilizado consiste en postular la existencia de una tabla extensible (abstracta, no como las **tabl**as del lenguaje; como las tablas de los libros, p. ej. una tabla de pesos atómicos) 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 90. Tabla de palabras y sus frecuencias de aparición

En esta tabla, las palabras estan ordenadas, y van insertándose (en la posición correcta) las nuevas a medida que se encuentran; inicialmente, está vacía.

El programa (incompleto, ya que falta la implementación del **modulo** *frecuencias*) supone que tratamos la tabla mediante las siguientes operaciones:

- *valor(p)*, donde *p* es una palabra, es la frecuencia de aparición de la palabra, que se supone encontrada al menos una vez;
- *está(p)* es una **condicion** que nos dice si una palabra nueva aparece por primera vez (~ *está(p)*) o no (*está(p)*) en la tabla;
- *cabenmás* indica si se ha llenado la tabla (suponiendo, lo cual, como se verá, puede ser *falso*) que su capacidad está acotada;
- *davalor p,n* distingue entre dos casos: si la palabra *p* ya está en la tabla, cambia su frecuencia, que pasa a valer *n*; si no está (y *cabenmás*), la mete en la tabla, con frecuencia *n*; por último,
- *valores* es una **sucesion** que produce la lista ordenada de las palabras y sus frecuencias (a ser utilizada al final del proceso).

Se notará que, con esta descripción, no hacemos suposición alguna sobre la forma en que se representará la tabla, sino que sólo anunciamos cuáles son sus propiedades. Esto es suficiente para entender el siguiente programa:

```
programa analiza_frecuencias es

  tipo clave es tira(10);

  modulo frecuencias es
    funcion valor(const c: clave): entero;
    condicion está(const c: clave);
    condicion cabenmás;
    accion davalor(const c: clave; i: entero);
    tipo par es tupla c: clave; n: entero; fin;
    sucesion valores: par;
  fin frecuencias;

  -- aquí debería figurar la parte de implementación
  -- del modulo analiza_frecuencias;
  -- las dos figuras que siguen son versiones correctas de tal implementación

  usa frecuencias;
  -- permite utilizar las entidades definidas en el modulo Analiza_frecuencias

  var ch: caracter; (* último carácter leído *)

  accion lee_tira(var c: clave) haz
    c ← ""; mientras ~ fdf ∧ ch = ' ' haz lee ch; fin;
    si ~ 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 si ~ todo_bien vv c = "";

    si
      □ está(c) ⇒ davalor c,valor(c) + 1; -- incrementa frecuencia
      □ cabenmás ⇒ davalor c,1; -- crea frecuencia
      □ otros ⇒ todo_bien ← falso; -- no caben más
        escribe_linea "Error en la tabla de frecuencias";
    fin si;

  fin itera;
```

```
  si todo_bien entonces
    para p en valores haz
      escribe_linea "La palabra '",p.c,"' aparece ",p.n," veces.";
    fin para;
  fin si;

fin programa;
```

Algoritmo 48. (Incompleto) Frecuencia de aparición de palabras en una frase

Distintos modelos o ***implementaciones*** pueden servir para los objetivos que nos hemos propuesto. La versión que sigue utiliza un árbol binario: en cada nodo se encuentra una palabra con su frecuencia asociada; el árbol está ordenado de modo que la palabra de todo nodo es mayor que toda palabra de su subárbol izquierdo, y mayor que toda palabra de su subárbol derecho (lo cual [vease el Algoritmo 45 en la página 203] nos permite obtener la lista ordenada mediante un recorrido en inorden [Cfr. el Algoritmo 46 en la página 205]). La condición *cabenmás* es siempre *cierto*, ya que (aparte de las limitaciones impuestas por la memoria disponible) no hay límite superior al número de nodos de un árbol implementado mediante **nombres**. La parte ejecutable de la implementación establece la condición inicial de tabla vacía (*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
          si
            □ cl = c ⇒ vale n;
            □ cl < c ⇒ act ← sig;
            □ cl > c ⇒ act ← ant;
          fin si;
        fin con;
      hastaque act = nulo;
    fin si;
    (* Error : clave errónea *)
```

```
fin valor;

condicion está(const c: clave) es
  var act: arbol;
haz
  si raiz ≠ nulo entonces
    act ← raiz;
    repite
      con act↑ haz
        si
          □ cl = c ⇒ vale cierto;
          □ cl < c ⇒ act ← sig;
          □ cl > c ⇒ act ← ant;
        fin si;
      fin con;
    hastaque act = nulo;
  fin si;
  vale falso;
fin está;

condicion cabenmás 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 si;
        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 davalor;

sucesion valores: par es
```

```
    sucesion 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 versión implementa el árbol en forma de **tabla**. substituyendo los **nombres** por (sub)índices de la tabla. El papel que jugaba **nulo** en la anterior figura lo realiza aquí el índice 0; en este caso, *cabenmás* no es siempre *cierto*, sino que depende del número de palabras encontradas: el espacio para un nuevo nodo se obtiene de la primera posición *libre* de la tabla (en vez de mediante la operación *crea*), que puede no existir (*libre* > *100*). En lo demás, 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
          si
            □ cl = c ⇒ vale n;
            □ cl < c ⇒ act ← sig;
            □ cl > c ⇒ act ← ant;
          fin si;
        fin con;
      hastaque act = 0;
    fin si;
    (* Error : clave errónea *)
  fin valor;

  condicion está(const c: clave) es
    var act: [0..100];
  haz
    si libre > 1 entonces
      act ← 1;
      repite
        con t[act] haz
          si
            □ cl = c ⇒ vale cierto;
            □ cl < c ⇒ act ← sig;
            □ cl > c ⇒ act ← ant;
          fin si;
        fin con;
      hastaque act = 0;
    fin si;
    vale falso;
  fin está;

  condicion cabenmás 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 si;
        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 davalor;

sucesion valores: par es
  sucesion 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 además una implementación sin **nombres** de un árbol).

El siguiente ejemplo, algo artificioso, intenta mostrar las posibilidades de protección que ofrecen los **modulos**.

Acceso controlado a variables

Supongamos esencial para el correcto funcionamiento de un programa que una variable entera *n* contenga sólo valores impares. Ello puede conseguirse de varios modos:

- En primer lugar, puede añadirse una instrucción de invocación del estilo de *verifica n;* después de cualquier potencial alteración de *n*, suponiendo que *verifica* es una acción que controla si efectivamente el valor de *n* es impar. Esto tiene el inconveniente de ser tedioso al obligar a escribir mucho; además, puede ser que se olvide algún punto en el que "*n* es impar" debe verificarse.

- Otra alternativa es diseñar subprogramas que realicen las operaciones de inspección y modificación sobre *n*, y evitar sistemáticamente cualquier referencia directa a *n*. Esta solución es más cómoda, pero mantiene el inconveniente de que, si por inadvertencia en algún 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 desearía utilizar la estructura del segundo, pero prohibir de algún modo el acceso a *n*. Para ello, se utiliza un **modulo**, cuya función 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 capítulo 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 "más externo" de la pila; por último,
- *Vacía* y *llena* son condiciones que indican el estado de la pila

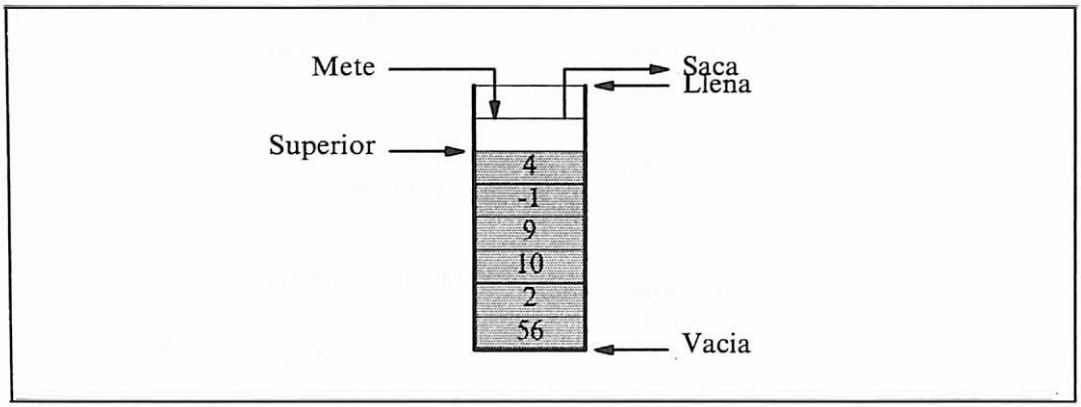


Figura 91. Una pila de enteros: El sombreado representa la parte utilizada.

Damos una definición y dos implementaciones de las pilas; entre estas últimas, una es no acotada (tubo con capacidad infinita o infinitamente largo) y se realiza con **nombres** y otra es acotada y utiliza **tablas** (compárese con las implementaciones de la tabla de frecuencias presentadas más arriba).

```
modulo pila es
  accion mete(e: entero);
  accion saca;
  funcion superior: entero;
  condicion vacia;
  condicion llena;
fin pila;
```

Algoritmo 52. Definición 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 vacía entonces escribe_linea "Error"; sino p ← p↑.p; fin;
  fin saca;

  funcion superior: entero;
    si vacía entonces escribe_linea "Error"; sino vale p↑.e; fin;
  fin superior;

  condicion vacía haz vale p = nulo; fin;
  condicion llena haz vale falso; fin;

haz
  p ← nulo;
fin pila;
```

Algoritmo 53. ***Implementación 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 vacía entonces escribe_linea "Error"; sino i ← i - 1; fin;
 fin saca;

 funcion superior: entero;
   si vacía entonces escribe_linea "Error"; sino vale p[i]; fin;
 fin superior;

 condicion vacía haz vale i = 0; fin;
 condicion llena haz vale i = 100; fin;

haz
  i ← 0;
fin pila;
```

Algoritmo 54. Implementación de una pila de enteros mediante tablas

Apéndice 1: Lista de reglas sintácticas

argumento = *expresión*

argumentos = *argumento* {,*argumento*}

bloque =
  [**es**
    {*declaración*}]
  haz
    instrucción
    {*instrucción*}
  fin [*identificador*];

cabecera =
    accion *identificador* ["("*lista_de_parámetros*")"]
  | **condicion** *identificador* ["("*lista_de_parámetros*")"]
  | **funcion** *identificador* ["("*lista_de_parámetros*")"]: *identificador*
  | **sucesion** *identificador* ["("*lista_de_parámetros*")"]: *identificador*

campo = *identificador*

carácter =
  letra | *dígito* | *carácter_especial* | *otros_caracteres* | *%espacio en blanco%*

carácter_constante = *'caracter'*

carácter_especial =
  "(" | ")" | "[" | "]" | "{" | "}" | + | - | * | ^ | v | ~
  | / | "\" | " | ' | ↑ | _ | < | > | : | ; | . | , | "="

condición = *expresión*

conjunto = "{" *expresión* {, *expresión*} "}"

declaración =
  declaración_de_módulo
  | *declaración_de_objeto*
  | *declaración_de_subprograma*
  | *declaración_de_tipo*

declaración_de_constante = **const** *identificador* "=" *expresión_constante;*

declaración_de_módulo = *definición_de_módulo* | *implementación_de_módulo*

declaración_de_objeto =
  declaración_de_constante
  | *declaración_de_variable*

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

declaración_de_subprograma = *cabecera bloque* | *cabecera;*

declaración_de_tipo =
  tipo *identificador* [**es** [**nombre**] *tipo*];

definición = *declaración_de_objeto*
  | *declaración_de_tipo*
  | *cabecera ;*
  | *definición_de_módulo*

definición_de_módulo =
  modulo *identificador* **es**
    definición
    {*definición*}
  fin [*identificador*];

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

discriminante = *identificador*

entero_sin_signo = *digito*{*digito*}

existencial = **existe** *variable* **en** *iteración* [**talque** *condición*]

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

expresión = *relación* { ∧ *relación* }
  | *relación* { ∨ *relación* }
  | *relación* { ∧∧ *relación* }
  | *relación* { ∨∨ *relación* }

expresion_constante = *expresión*

expresion_simple = [+|-] *término* { *operador_aditivo término* }

```
factor = número_sin_signo
  | tira
  | carácter_constante
  | conjunto
  | existencial
  | variable
  | selección_de_tira
  | invocación_de_función
  | invocación_de_condición
  | conversión_de_tipo
  | "(" expresión ")"
  | ~ factor

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

implementación_de_módulo =
  modulo implementa identificador es
    {declaración}
  [haz
    instrucción
    {instrucción}]
  fin [identificador];

instrucción =
  instrucción_de_asignación
  | instrucción_de_invocación
  | instrucción_nada
  | instrucción_vale
  | instrucción_acaba
  | instrucción_produce
  | instrucción_sal
  | instrucción_si
  | instrucción_repite
  | instrucción_mientras
  | instrucción_itera
  | instrucción_para
  | instrucción_con

instrucción_acaba = acaba;

instrucción_con =
  con variable_tupla {,variable_tupla} haz
    instrucción
    {instrucción}
  fin [con];

instrucción_de_asignación = variable ← expresión;

instrucción_de_invocacion = identificador [argumentos];
```

```
instrucción_itera
  itera
    instrucción
    {instrucción}
  fin [itera];

instrucción_mientras =
  mientras condición haz
    instrucción
    {instrucción}
  fin [mientras];

instrucción_nada = nada;

instrucción_para =
  para variable en iteración [talque condición] haz
    instrucción
    {instrucción}
  fin [para];

instrucción_produce = produce expresión;

instrucción_repite =
  repite
    instrucción
    {instrucción}
  hastaque condición;

instrucción_sal = sal [identificador] [si condición];

instrucción_si =
  si
    □ condición ⇒
      instrucción
      {instrucción}
    {□ condición ⇒
      instrucción
      {instrucción}}
    [□ otros ⇒
      instrucción
      {instrucción}]
  fin [si];
```

```
instruccion_si =
 si expresion es
   □ lista_de_valores ⇒
     instruccion
     {instruccion}
   {□ lista_de_valores ⇒
     instruccion
     {instruccion}}
   /□ otros ⇒
     instruccion
     {instruccion}]
 fin /si/;

instrucción_si =
 si condición entonces
   instrucción
   {instrucción}
  /sino
   instrucción
   {instrucción}]
 fin /si/;

instrucción_vale = vale expresión;

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

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

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

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

lista_de_identificadores = identificador {,identificador}

lista_de_parámetros = parámetros {; parámetros}

lista_de_valores = valor_o_rango { , valor_o_rango }

número_sin_signo = entero_sin_signo | real_sin_signo

operador_aditivo = + | -

operador_multiplicativo = * | / | div | mod

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

otros_caracteres = %otros caracteres, segun la versión%

parámetros = parámetros_por_copia
  | parámetros_por_nombre
  | parámetros_por_subprograma
```

```
parámetros_por_constante = const lista_de_identificadores: identificador;

parámetros_por_copia ≐ lista_de_identificadores: identificador;

parámetros_por_nombre = parámetros_por_variable | parámetros_por_constante

parámetros_por_subprograma = cabecera

parámetros_por_variable = var lista_de_identificadores: identificador;

parte_decimal = entero_sin_signo

parte_entera = entero_sin_signo

parte_variante =
  si discriminante: tipo_del_discriminante es
    □ valor_o_rango { , valor_o_rango} ⇒
      {campo {,campo}: tipo;}
      [parte_variante]
    {□ valor_o_rango { , valor_o_rango} ⇒
      {campo {,campo}: tipo;}}
      [parte_variante]
    [□ otros ⇒
      {campo {,campo}: tipo;}]
      [parte_variante]
  fin [si];

programa =
  programa identificador es
    declaración
    {declaración}
  haz
    instrucción
    {instrucción}
  fin programa;

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

relación = expresión_simple [ operador_relacional expresión_simple ]

selección_de_tira = variable "[" expresión [ ..expresión ] "]"

selector = ↑
  | "[" expresión {,expresión} "]"
  | . identificador

simbolo_especial = cáracter_especial | ← | ≤ | ≠ | ≥ | ⇒ | .. | □

término = factor { operador_multiplicativo factor }

tira = "{caracter}"
```

```
tipo = identificador
  | TIRA "(" expresión_constante ")"
  | tipo_enumerado
  | tipo_subrango
  | tipo_tabla
  | tipo_tupla
  | tipo_conjunto
  | tipo_fila

tipo_base = identificador | tipo_enumerado | tipo_subrango

tipo_conjunto = conjunto de tipo_base

tipo_del_discriminante = identificador

tipo_elemento = tipo

tipo_enumerado = tipo_enumerado_no_ordenado
  | tipo_enumerado_ordenado
  | tipo_enumerado_ciclico

tipo_enumerado_ciclico = "(" lista_de_identificadores ")" ciclico

tipo_enumerado_no_ordenado = "{" lista_de_identificadores "}"

tipo_enumerado_ordenado = "(" lista_de_identificadores ")"

tipo_indice = identificador | tipo_enumerado | tipo_subrango

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

tipo_tabla = tabla tipo_indice {,tipo_indice} de tipo_elemento

tipo_tupla = tupla_fija | tupla_con_variantes

tupla_con_variantes =
  tupla
    {campo {,campo}: tipo;}
    parte_variante
  fin [tupla];

tupla_fija =
  tupla
    campo {,campo} : tipo;
    {campo {,campo} : tipo;}
  fin [tupla]

valor_o_rango = expresión_constante [ .. expresión_constante]

variable = identificador {selector}

variable_tupla = variable
```

Apéndice 2: El compilador UBL/CMS versión 0.2

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 método de "bootstrapping". El compilador consta de dos fases: en la primera, se realiza el análisis sintáctico y semántico y se genera un listado de compilación y un fichero que contiene código (PCODE) para una versión modificada del "Stack Computer" descrito en la información distribuida con Pascal P4; en la segunda, se ensambla el código intermedio (PCODE) en formato compatible con los cargadores de los sistemas operativos para IBM/370 y similares. Ese código se ejecuta con la ayuda de un intérprete y de una librería de programas para la ejecución.

El siguiente diagrama muestra la estructura del compilador UBL/CMS y de los ficheros generados.

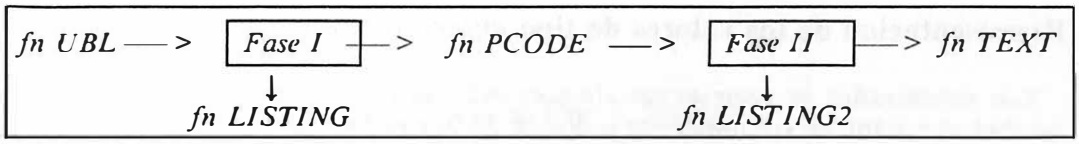


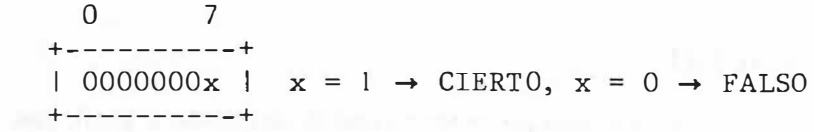
Figura 92. ***Estructura y funcionamiento del compilador UBL/CMS:*** los ficheros de tipo PCODE y LISTING2 son destruidos después de la compilación (a menos que se especifique lo contrario): así, el compilador produce un LISTING y un TEXT, como sucede el la mayoría de los demás lenguajes.

Representación interna de los datos

La representación interna de los datos se ha escogido para facilitar la compilación, y en algunos casos (p. ej., en los valores enumerados y en las tiras de caracteres) supone un claro desperdicio de memoria.

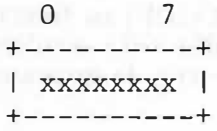
Representación de los valores de tipo *logico*

Se representan utilizando un byte: el bit situado más a la derecha es 1 si el valor es *cierto*, y 0 si es *falso*; los demás bits deben ser siempre 0.



Representación de los valores de tipo *caracter*

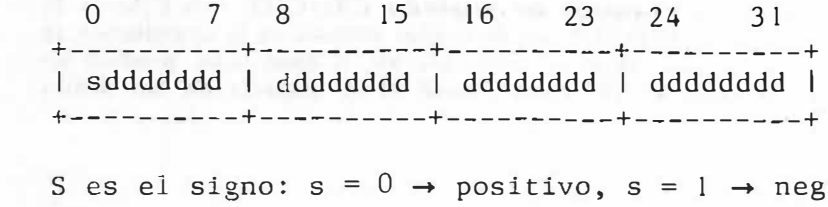
Se representan en el código EBCDIC utilizando un byte.



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).



Los dígitos D representan el número en binario, si es positivo, o el complemento lógico de su valor absoluto, más uno, si es negativo: por ejemplo, el entero -5 se representa del siguiente modo:

```
S = 1, ya que -5 es negativo; para calcular los dígitos 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 último añadimos el bit S de signo a la izquierda, se
obtiene la representación buscada:

  11111111 11111111 11111111 11111011
```

Los límites a los valores enteros impuestos por esta representación 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.

Representación 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 normalización en base 16:

```
  0       7   8       15   16      23   24      31
+----------+----------+----------+----------+
| Sseeeeee | mmmmmmmm | mmmmmmmm | mmmmmmmm |
+----------+----------+----------+----------+

S: signo del numero
s: signo del exponente
eeeeee: exponente (potencia de 16)
mmmm... : mantisa (el primer m, distinto de 0)
```

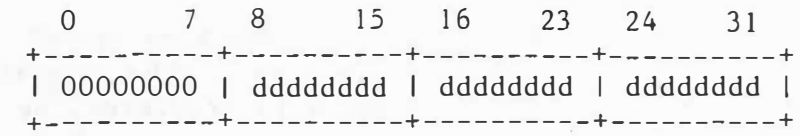
Con esta representación, se obtiene una precisión algo superior a los seis dígitos decimales, y un rango de exponentes situado entre -74 y 76, aproximadamente.

Los reales se alinean a frontera de 4 bytes.

No se implementan reales de precisión doble (REAL*8 o DOUBLE PRECISION de FORTRAN) ni cuádruple (REAL*16).

Representación 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.

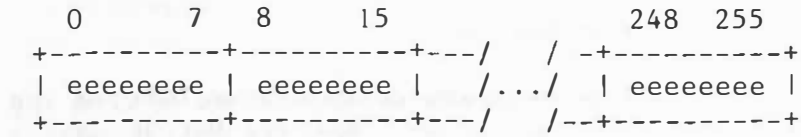


ddd... es la dirección del objeto nombrado.

Los nombres se alinean a frontera de 4 bytes.

Representación 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: así puede considerarse que todos los conjuntos son de enteros.



e = 1 si el elemento está en el conjunto
e = 0 si el elemento no está en el conjunto

Los ordinales máximo y mínimo 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 está en el conjunto, y 0 si no está (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.

Representación 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 alineación correcta.

Las tiras de caracteres se representan como una tabla de caracteres cuya longitud es la máxima 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 más, también predefinidas y auxiliares, llamadas *entrada2* (en modo de entrada) y *salida2* (en modo de salida). Las características 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 instrucción de CMS FILEDEF.

En la versión 0.2 algunas operaciones sofisticadas de entrada y salida están en fase de implementación.

Instrucciones '%'

El compilador admite instrucciones que comienzan con el símbolo "%" para controlar la apariencia del listado, cuáles columnas son significativas para la compilación, etc.

Pueden utilizarse las siguientes:

% PAGINA provoca un salto de página en el listado: la siguiente línea sera la primera de la siguiente página.

% TITULO tira de caracteres los caracteres siguientes al blanco que sigue a la palabra "titulo" pasan a imprimirse en la primera línea de cada página del listado. La instrucción fuerza también un cambio de página.

% 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 instrucción "incluye", se hallase el texto correspondiente al fichero de nombre *filename* y tipo UBL.

Si la instrucción es correcta, no aparece en el listado.

Utilización: instrucciones de CMS

Compilación

Los ficheros que contienen programas en UBL deben ser de longitud fija y de 80 caracteres por línea. Se compilan mediante la instrucción (EXEC) de CMS UBL, cuyo formato se describe a continuación:

```
        -           -           -          - -              -
UBL     | filename  | filetype  | filemode | |  ( opciones  |
        | ?         | UBL       | *        | |              |
        | *         -           -          - -              |
        | (?                                                 |
        -                                                    -

   opciones: CAStellano | CATala
             Source | NOSource
             HOld | NOHold | PCode
             UNder | NOUNder | SUBR
             OVer | NOOVer | NEGR
             DIsk | PRint | NOPRint
             OBJect | NOOBJect
             Debug | NODebug | ND
```

Figura 93. ***Formato de la instrucción UBL:*** la parte de las opciones que aparece en minúsculas es opcional.

"UBL ?" proporciona información interactiva sobre el formato y funcionamiento de la instrucción UBL; "UBL (?" lo hace con las opciones de compilación.

A continuación 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 método es acumulativo: las opciones de una instrucción '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 está escrito el programa que se desea compilar. Debe especificarse esta opción; de lo contrario, se supone que el programa está en inglés.

- SOURCE | NOSOURCE decide si el listado producido por el compilador contendrá una reproducción 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 traducción 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 ejecución 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 línea vertical y por arriba mediante una línea de escala. A la izquierda de la línea vertical aparecen numeraciones de las líneas del programa (bajo el título "Linea") y de las instrucciones (bajo el título "Sentencia"). Las instrucciones "%" no aparecen en el listado, si son correctas. Cada línea con errores está seguida de indicaciones de las posiciones en las que ocurren los errores y de un número que identifica el error, así como de líneas 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 símbolo léxico siguiente a aquel en el que realmente ocurren; también es posible la aparición de errores en cascada.

Ejecución

Para ejecutar un programa compilado en UBL, se utilizará la instrucción RUBL:

```
RUBL filename
```

donde "filename" es el nombre del fichero que se compiló. No debe utilizarse la secuencia habitual de instrucciones LOAD y START, ni RUN. Tampoco es necesario hacer ningun GLOBAL, aunque quiza sí algun FILEDEF (antes de RUBL) si los ficheros que se usan no son la terminal.

Escritura de programas en terminales de tipo 3278

En el texto figuran algunos símbolos que no se encuentran en las terminales de tipo 3278, normalmente utilizadas con el sistema CMS; el compilador de UBL acepta caracteres que substituyen a estos símbolos, según la siguiente tabla:

El símbolo	*se escribe*
←	<-
⇒	=>
□	[ ]
∨	|
∧	&
~	¬
≤	<=
≥	>=
≠	<>
↑	@

Figura 94. ***Escritura de símbolos que no están en las terminales***

Además, las terminales no soportan acentos, ni letra negrita ni cursiva, por lo que deberán introducirse los programas sin este tipo de resalte.

Se sugiere que la indentación de las líneas sea de dos espacios.

Bibliografía

Referencias recomendadas

1. 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.

2. 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.

3. 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.

4. Dijkstra, E.W.: *A Discipline of Programming.* Prentice-Hall, 1976.

   Otra visión, muy formalizada, de la demostración formal de programas.

5. Dijkstra, E.W.: *Selected Writings on Computing: A Personal Perspective.* Springer-Verlag, 1982.

   Interesante, entre otras cosas, por las opiniones que expresa.

Referencias citadas en la introducción

[ADA 83] *Reference Manual for the Ada Programming Languaje*, ANSI/MIL-STD-1815A-1983.

[BlaA 84] Blasco, J. M.; Alonso, G.: *UBL (Lenguaje de la Universidad de Barcelona): un lenguaje para la enseñanza de la programación en castellano.*. Comunicación presentada en las "Jornadas sobre Informática y Educación en la enseñanza básica y media", Madrid, 26-28 Noviembre 1984. También en *Informática y Escuela*, Fundesco, 1985.

[Dijk 82] Dijkstra, E. W.: *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag 1982.

[Dijk 83] Orejas, F.; Llamosi, A.: *Tot fent el café amb el professor Dijkstra*, (Ciència), Noviembre 1983: 46-51.

[Lisk 77] Liskov, B.;Snyder, A.;Atkinson, R.;Schaffert, C.: *Abstraction Mechanisms in CLU*. CACM 20, no. 8, Agosto 1977.

[PBot 83] Botella, P.;Orejas, F. *Merli: Report Preliminar*, Departamento de Programación. Facultad de Informática de Barcelona, 1983.

[PBot 84] Botella, P.: *Reflexiones Pedagógicas en torno a la Enseñanza de la Programación*. comunicación presentada en el 1er. Simposio sobre Informática y Educación, celebrado en Tucumán (Argentina).

[Shaw 77] Shaw, M.;Wulf, W. A.;London, R. L.: *Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators*, CACM 20, no. 8, Agosto 1977.

[Tub 84a] Tubau, E.;Sopena, J. M.;Blasco, J. M.;Sebastian, N.; Alonso, G.: *Valoración pedagógica de las opciones lingüísticas del lenguaje experimental UBL en la enseñanza de la programación*, comunicación presentada en las "Primeras Jornadas Nacionales sobre Informática en la Enseñanza" (Barbastro, 11-14 de Julio 1984)

[Tub 84b] Tubau, E.: *Psicologia del Software: Factors Congnitius en l'aprenentatge i utilització de llenguatges de Programació*, Tesis de Licenciatura presentada en la Facultad de Psicologia de la Universidad de Barcelona, Septiembre de 1984.

[Tub 84c] Tubau, E.;Sopena, J. M.;Blasco, J. M.;Sebastian, N.; Alonso, G.: *Aprendizaje de lenguajes de programación en la propia lengua: experiencia de valoración comparativa*, comunicación presentada en las "Jornadas Sobre 'Informática y Educación en la enseñanza Básica y Media'" , Madrid, 26-28 Noviembre 1984. También en *Informática y Escuela*, Fundesco, 1985.

[Wirth 82] Wirth, N.: *Programming in Modula-2*, Springer-Verlag, Berlin 1982.

Índice Alfabético

A

abstracción 49
    niveles de
acceso aleatorio 167
acceso secuencial 167
accion 48
acción 9
    predefinida 169
        conecta 169
        crea 196
        desconecta 169
        escribe 170
        escribe_linea 173
        lee 170
        lee_linea 173
        libera 196
        obten 169
        pon 169
activación ver subprograma, 88
algoritmo 7, 2
alias 75
anuncio 72, ver definicion de subprograma
apuntador ver **nombre**
árbol
    binario 201
    de invocaciones 184
argumento 74
    lista de 77
asignación 10, 4, 5
    operador de 10
    parte derecha 10
    parte izquierda 10

B

backtracking 147
buffer ver ventana, **fila**

C

campo 158, ver **tupla**, 161
    discriminante 163
    fijo 163
    variante 163
        accesibilidad y existencia 163
comentario 33
    sobredocumentación 34
    subdocumentación 34
compatible 54
compilación 7
    errores de 7
    listado de 7
condición 11, 19
condicion 48, 51
conjunto ver tipo **conjunto**
    de caracteres o "character set" 21
        contiguidad 21
    sintaxis de un 119
    vacio 121
constante 10, 14, 99
contiguidad de los digitos 103
conversión 55
    de tipo 55, 100
        a enumerado 100
        a *tira* 55
corrección de un programa 7

D

datos 7
declaración 4, 5
    de **accion** 51
    de **condicion** 51
    de **funcion** 79
    de **modulo** 207
    de objetos 13, 39
    de subprograma 72, 77
    de **tipo** 102
    de **tipo** para *tiras* 81
    local 71, ver también entidad local

usa 208
definición 207
de subprograma 72, 185
parte de 207
depuración 8
discriminante ver campo discriminante
diseño descendente 43
dispositivo de almacenamiento externo 167
división de un problema 44
documentación ver Comentario

E

efecto 9
ejecución 7, 4, 9
secuencial 9
orden de la 9
entidad 71
local 71
accesibilidad y existencia 71
entrada y salida 12, 107
formatos de 114
interactiva 115, 116
enumerado ver tipo enumerado, 102
error 7
de compilación 7
de truncación 54
lógico 8
escritura 5, 168
modo de - para una **fila**
estado 9
estilo 27, 35
evaluación 67
existencial 89
expresión 67, 11, 63

F

fichero 167
fusión 172
ubicación 167, 169
fila 167, ver tipo **fila**
de caracteres 172
ventana o 'buffer' de una 168, ver seleccion
fin de fila 171, ver funcion predefinida, **fila**
convenciones de 171
fin de línea 173, ver funcion predefinida
carácter de 173
formato de entrada y salida ver entrada y salida
función 79
predefinida 80
abs 80
fdf 170
fdl 173
impar 81
long 55
ordinal 100
pred 81, 100
suc 81, 100
trunc 81

G

grafo de invocaciones ver árbol de invocaciones

I

identificador 10
predefinido 32
reservado 32
implementación 185
de un subprograma
parte de - de un **modulo** 207
indentación 35, 5
nivel de 35
indice 54
familias indexadas 125
inicialización 16
inspección 10, 168
instancia 183
instrucción 7, 107
acaba 48, 52
con 165
ámbito de la 165
de asignación 38, 10
de invocación 51
itera 108, 109
mientras 107, 108
nada 44, 46
para 88
produce 87, 91
repite 38, 11
sal 108, 109
si -- forma general 44, 46
si -- forma simplificada 37
si -- forma factorizada 112
subordinada 35
vale 49, 48, 80
interpretación 7
iteración 11, 5

L

lectura 4, 168
  modo de - para una **fila** 168
lista 198, 198
literal 10
local ver declaración local
longitud ver tira

M

matriz 135, 144
metalenguaje 27
metasimbolo 27
  terminal 28
  no terminal 28
modificación 10, 168
modo de acceso a un fichero ver lectura,escritura
modulo 207
  parte de definición 207
  parte de implementación 207

N

nombre 195
nulo 195
números complejos 157

O

objeto 9
  constante 10
  declaración de 13
  inspección de un 10
  modificación de un 10
  nombre de un 10
  tipo de un 10
  valor de un 10
  variable 10
operador
  aritmético 63
  conjuntista 119
    diferencia 121
    inclusión 120
    intersección 121
    pertenencia 120
    unión 121
  de concatenación 54
  lógico 19
    condicional 65
    relacional 20
  prioridad de un 67
  sobre **filas** 168
  sobre **tablas** 130, ver seleccion
ordenación lexicográfica ver comparacion de tiras

P

palabra clave 32
parámetros 71, 73, 80
  lista de 74
  por constante 75
  por copia 74
  por subprograma 179
  por variable 74, 168
periférico ver dispositivo de almacenamiento externo
pointer ver **nombre**
problema
  análisis del 8
producto cartesiano 157
programa 7, 2
  escritura de un 8
  forma de un 14
  prueba de un 8

R

reconocimiento de nombres 72
  esconder 72, 166, 208
recursividad 183
  directa 185, 197
  indirecta 185, 197
  mutua 185
  subprograma recursivo 183
  tipo recursivo 197
refinamiento 43, ver diseño descendente, 43, 44
  progresivo 43

S

simbolo 30
selección 12, 5, 161, 168, 169
  de campos de una **tupla** 158
  de elementos en una **tabla** 129, 130
  de un carácter de una *tira* 54
sintaxis 27
  de la lista de argumentos 77
  de la lista de parámetros 74
  de la selección 130
  de las declaraciones
    de **accion** 51

de **condicion** 51
de **modulo** 207
de objeto 13
de subprograma 77
de tipos *tira* 81
de tipos **nombre** 195
usa 208
de las expresiones 68
de las instrucciones
con 165
de asignación 38, 10
de invocación 77, 51
itera 108
mientras 108
nada 46
para 89
produce 91
repite 38, 11
sal 109
si -- forma general 46
si -- forma simplificada 37, 12
si -- forma factorizada 112
vale 49, 80
de los parámetros por subprograma 179
de los tipos 102
conjunto 120
enumerados 102
fila 168
subrango 103
tabla 129
tupla 161
de un conjunto 119
de un existencial 90
de un **programa** 39
regla sintáctica 27
sinónimo 74, ver parametro por copia
subíndice ver índice
subprograma 52
activación de un 52
actual 179
como parámetro 179
formal 179
genérico 179
terminación de un 52, 88
subrango ver tipo subrango
subtira 54
límite inferior 54
límite superior 54
sucesion 87
activación 88
continuación 88
predefinida 88
asc 88, 100
desc 88, 100
suspensión 88
terminación 88

T

tabla 125, ver tipo tabla
texto 172, ver **fila**, fila de caracteres
tipo 10, 13, 102
caracter 13, 102
conjunto 119, 119, 120
tipo base 120
entero 13, 102
enumerado 99, 102
cíclico 100
no ordenado 100
ordenado 100
estructurado 53, 159, 167
fila 167, 168
heterogeneo 159
homogeneo 159
logico 63, 65, 102
nombre 195
real 63
recursivo 197
simple 159
subrango 103, 99, 103
tabla 129
tipo índice 130
tira 53, 81
tupla 161
con variantes 162, 163
tira 53, 81
comparación de 55
de caracteres 32
longitud de una 53, 55
longitud actual 53
longitud máxima 53
vacía 32, 53
truncación 54
tupla 157
campos de una 158
con variantes 162, 164
estructura de una 159

U

unión disjunta 157, 164

V

valor 10
indefinido de un objeto 14
variable 10, 4, 5, 13
vector 128, ver **tabla**
búsqueda de elementos 153
componentes 130
elementos 130

ordenación de un 150
método de la burbuja 151
ventana 168, ver **fila**
visible 208
vocabulario 30

CENTRE D'INFORMÀTICA · UNIVERSITAT DE BARCELONA

PPU · Promociones Publicaciones Universitarias