Pseudolenguaje 3.2

E.T.S.I. Telecomunicación

Dpto. Lenguajes y Ciencias de la Computación

lunes 25 de septiembre de 2002

Contenido

1. Elementos léxicos

Un *programa* puede verse como una gran cadena de caracteres del conjunto estándar de la máquina (ASCII, Latino 1 o Unicode por ejemplo). Lo primero que hace un compilador para traducir un programa de alto nivel es agrupar los caracteres del programa en *unidades léxicas* básicas o *tokens*. Combinando estas unidades léxicas de acuerdo a unas reglas *sintácticas* obtenemos las *sentencias*, que serán *declarativas* o *ejecutables*. A continuación se enumeran los elementos léxicos típicos de los lenguajes de tipo imperativo.

Palabras clave o *reservadas*: cadenas de letras con sentido predefinido en el lenguaje. En nuestro pseudo-lenguaje las escribiremos con todas sus letras en mayúsculas.

Literal: datos constantes sin nombre. Pueden ser numéricos como `2`, `-5`, `2.3`, `3.103e-23` o de caracteres. Estos últimos deben ir entre comillas simples si constan de un solo carácter, `'a'`, `'\n'`; o dobles si son cadenas, `"Hola, ¿qué tal?"`.

Identificador: cadena de caracteres alfanuméricos (letras y dígitos) que comienza con una letra y que no es palabra reservada. En este sentido el símbolo de subrayado '_' se considera una letra. Se utilizan para que el programador dé nombre a las distintas entidades que contiene el programa.

Operadores: símbolos o combinaciones cortas de símbolos que representan operaciones aritmético-lógicas. Los que constan sólo de letras pueden considerarse también palabras reservadas. Distinguiremos los siguientes operadores[1]:

Aritméticos: **`+ - * / DIV MOD`**

Relacionales y de igualdad: **`< > <= >= != ==`**

Lógicos: **`Y O NO`**

De accesibilidad: **`. [] ^`**

Delimitadores: resto de símbolos separadores (`<SP> <TAB> <NL> <RC>`), o de puntuación (**`, ; /* */ //`**).

Todas las palabras reservadas e identificadores deben separarse del resto de elementos léxicos con un símbolo separador al menos. En este sentido, los operadores `Y`, `O`, `NO`, `DIV` y `MOD` se consideran palabras reservadas.

2. Estructura general de un programa

Un (sub)programa consta fundamentalmente de una zona *declarativa* y de una zona *ejecutable*. En ambas zonas se puede insertar *comentarios* en cualquier punto del programa siempre y cuando no rompan las unidades léxicas. Estos comentarios son eliminados durante la traducción y no existen en el código máquina generado, son simplemente pequeñas notas de documentación para el propio programador que hacen más legible el programa.

En la zona declarativa se definen las características de utilización de todas las entidades no predefinidas en el lenguaje. Se escribe delante de la ejecutable. Las entidades a definir son las siguientes:

Constantes simbólicas: datos constantes con nombre. Se definen en la sección **`CONST`**. Adoptaremos el convenio de escribirlas con la primera letra por lo menos en mayúscula.

Tipos: determinan las características (dominio y operaciones) de los datos. Se definen en la sección **`TIPOS`**. Adoptaremos el convenio de nombrarlos comenzando con una **`T`** en mayúscula.

Variables: datos cuyo valor puede variar durante la ejecución de un programa. Se definen en la sección **`VAR`**. Adoptaremos el convenio de escribirlas con todas las letras en minúsculas.

(Sub)Programas: resolución computacional de un (sub)problema. Se definen fuera de las tres secciones anteriores. Adoptaremos el convenio de poner la primera letra de cada identificador de (sub)programa en mayúscula.

[1] El anexo I describe las *reglas de precedencia*, que determinan el orden implícito de aplicación de los operadores.

Clases: agrupación de datos relacionados entre sí y subprogramas con los que se pueden manipular de forma exclusiva. Se definen fuera de las tres secciones anteriores. Adoptaremos el convenio de nombrarlos comenzando con una **C** en mayúscula.

A excepción de los tipos predefinidos y algunos subprogramas predefinidos, todas estas entidades deben definirse antes de su utilización en su zona declarativa correspondiente. No obstante, no existe un orden preestablecido para las distintas secciones.

En la zona ejecutable se escribe en el orden adecuado las sentencias que realmente forman la resolución del problema. Todas las sentencias, tanto declarativas como ejecutables, terminan con el carácter de nueva línea o con un punto y coma.

En la parte ejecutable existen fundamentalmente las siguientes unidades sintácticas:

La *asignación*, sentencia de transferencia de datos por excelencia.

Decisiones y *repeticiones*, sentencias de control de flujo.

Llamadas a subprogramas, sentencias de control de flujo de *ida y vuelta*.

Expresiones: unidades sintácticas de tratamiento aritmético-lógico.

Con el fin de resaltar las distintas unidades sintácticas y algunas léxicas, en el presente documento se adopta el siguiente convenio de colores para escribir **ALGORITMO**s. Este convenio podrá utilizarse en cualquier documento que se suministre a los alumnos para ayudarles a "leer" un programa más fácilmente, por ejemplo, los expuestos en Internet. También podrá utilizarse con los entornos de programación que permitan configurarse así.

Las **PALABRAS RESERVADAS** y el símbolo de asignación **=** se escriben en negrita.

Los `/* comentarios */` se escriben en verde.

Las llamadas a `SubProgramas(<argumentos>)` se escriben en azul, pero no sus `<argumentos>`. Si los subprogramas son predefinidos[2] (no de biblioteca), además, se escriben con todas sus letras en mayúsculas, como por ejemplo, `ASIGNAR(<puntero>)`. (El nombre un algoritmo **<MiAlgoritmo>** se escribe en **azul** resaltado).

Las `<expresiones>`, a excepción de las llamadas a `Funciones()`, se escriben en el color por defecto. Esto incluye los `<operadores>`, las `<constantes>`, las `<variables>` y los `<argumentos>` de los subprogramas.

Los nombres de los `<Tipos>` y `<Clases>` se escriben en rojo.

A continuación se presenta el esquema general de un algoritmo.

```
ALGORITMO Identificador
/* comentarios multilínea:
   constan de más de una línea (estilo C) */
// comentarios de una sola línea (estilo C++)

CONST
   <declaraciones de constantes simbólicas>

TIPOS
   <declaraciones de tipo>
   <declaraciones de clases>

VAR
   <declaraciones de variables>
   <declaraciones de SubAlgoritmos>

INICIO
```

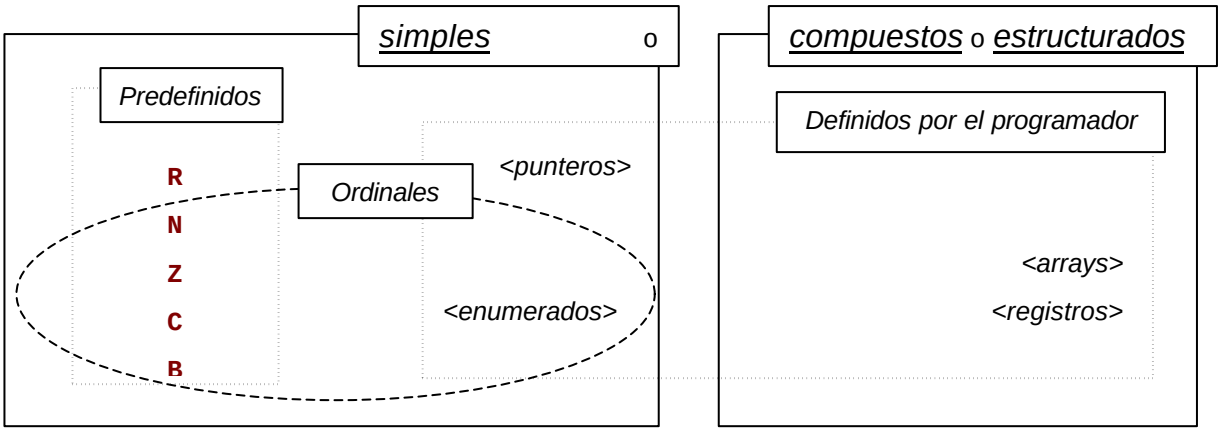
[2] Véase el anexo correspondiente.

```
  <sentencias ejecutables>

FIN
```

3. Tipos

Distinguiremos las siguientes categorías de tipos:



Consideramos *predefinidos* los tipos que ya tienen significado en pseudolenguaje sin necesidad de definirse. El anexo I describe los dominios y operaciones permitidas para cada uno de estos tipos.

Consideramos *ordinales* los tipos cuyos valores tienen sucesor único (a excepción del último) y predecesor único (a excepción del primero).

Conversiones de tipo *explícitas*: `<TipoDestino>(<expresión>)`

Conversiones de tipo *implícitas*: `Z` ← `N`

Se consideran tipos *compatibles* los que son iguales. Cuando sea necesario se promocionan los naturales a enteros (conversión implícita).

4. Expresiones

Una *expresión* es una combinación, según unas deteminadas reglas, de elementos que representan datos, los *operandos*, y conectivas que representan operaciones aritmético-lógicas, los *operadores*. La evaluación de una expresión produce un resultado, un valor, por lo que puede actuar a su vez como operando, dando lugar a las *subexpresiones*.

Como operando puede actuar:

- Una constante simbólica o literal.
- Una variable.
- Una llamada a una función[3].
- Una subexpresión.

Los operadores son los aritmético-lógicos ya enumerados. Los unarios se escriben delante de su operando, posiblemente encerrado entre paréntesis. Los binarios siguen la notación *infija*, en la que el operador se sitúa entre ambos operandos, los cuales pueden ir entre paréntesis. Todos los operandos son de un determinado tipo.

Además, deben seguirse unas reglas semánticas, a saber:

1. Los operandos de los operadores binarios deben ser compatibles entre sí.
2. Los operandos deben ser compatibles con los que exige cada operador.

Hay que tener en cuenta que el tipo de una (sub)expresión es el que devuelva su operador menos prioritario. También las llamadas a una función producen un resultado de un determinado tipo.

Ejemplos:

```
/* de tipo R */   23.0*x + Sqrt(y)/2.0 – 2.0*Pi   // R x, y, Pi=3.1416
```

[3] Ver en anexo II las funciones predefinidas.

```
/* de tipo Z */   (a DIV b) MOD 60 – ABS(c)       // Z a, b, c
/* de tipo C */   CAP(letra)                      // C letra
/* de tipo B */   vale O (a < b+3) Y (NO error)   // B vale, error
```

Desde el punto de vista semántico, hay que distinguir:

a) *Valores-L* (*L-values*): expresiones que siempre deben hacer referencia a una POSICIÓN DE MEMORIA, tales como variables simples, componentes de arrays, de registros o variables referenciadas mediante punteros.

b) *Valores-R* (*R-values*), que son expresiones que devuelven siempre un VALOR de un determinado tipo.

5. Entrada / Salida

Los tipos simples, a excepción de los lógicos y enumerados, y las cadenas de caracteres disponen de un subprograma predefinido para almacenar uno o varios valores leídos de teclado en uno o más valores-L:

```
Leer(<id_var1>, <id_var2>, <id_var3>, ...)
```

Una lectura sobre una cadena de caracteres leerá hasta el primer separador.

Las expresiones de tipo simple a excepción de las de tipo lógico y enumerado, así como las cadenas de caracteres, disponen de un subprograma predefinido para mostrar valores-R en pantalla:

```
Escribir(<expr1>, <expr2>, <expr3>, ...)
```

La entrada/salida de los valores numéricos es *formateada*, hay conversión del formato de almacenamiento interno de la máquina a cadenas de caracteres que forman el número en decimal o viceversa. La entrada/sadida de tipo carácter es *no formateada*, no hay tal conversión.

6. Constantes simbólicas y variables

En la notación BNF que utilizaremos, los caracteres **`terminales`** son los que van en negrita, los `<no terminales>` los que van entre angulos y los metasímbolos en texto normal.

BNF	En la sección **CONST**:
	`<Def_ConstSimbol> ::= <IdTipo> <IdCte>=<expr_cte> {, IdCte=<expr_cte>}`
	En la sección **VAR**:
	`<Def_Variable>  ::= <IdTipo> <id_var> [ = <expr_cte> ]`
	`{, <id_var> [ = <expr_cte> ] }`

Ejemplo	`ALGORITMO TiposBasicos`
	`VAR`
	`  N num1, num2`
	`  N num3 = 4  /* inicializacion en declaracion */`
	`  C car`
	`INICIO`
	`  Escribir(“Introducir carácter: ”)`
	`  Leer(car)`
	`  num2 = ORD(car)`
	`  num1 = num3 DIV num2`
	`  Escribir(“El resultado es: ”, num1)`
	`FIN`

Nota	Una variable comienza siempre con un valor indefinido. Como norma de estilo conviene no reutilizar variables para usos distintos. Se deben emplear identificadores autoexplicativos para recordar fácilmente el contenido de las constantes y variables.

7. Valores enumerados

Los valores *enumerados* son conjuntos de valores definidos explícitamente por el programador. Léxicamente, cada valor se representa mediante un identificador. El conjunto de valores de una definición constituye el dominio de un *tipo enumerado*. El orden de enumeración de los valores determina su orden para la aplicación de los operadores relacionales. Aparte de estos operadores, no se les puede aplicar ninguno de los operadores predefinidos.

BNF	En la sección **TIPOS**:
	`<Def_TipoEnum> ::= ENUM {<IdValor1>{, <IdValori>}} <IdTipoEnum>`
	En la sección **VAR**:
	`<Def_VarEnum>  ::= <IdTipoEnum> <id_var> [ = <ValorEnumerado> ]`
	`{, <id_var> [ = <ValorEnumerado> ] }`

Ejemplo	`ALGORITMO TiposEnumerados`
	`TIPOS`
	`  ENUM {Rojo, Amarillo, Azul, Verde} TParchis`
	`VAR`
	`  TParchis miparchis`
	`INICIO`
	`  miparchis = Rojo`
	`FIN`

Notas	Los enumerados no pueden leerse ni escribirse directamente. Se consideran tipos ordinales. Como norma de estilo, al igual que las constantes simbólicas, todos los valores enumerados comenzarán con mayúscula. No así las variables. No tienen operadores predefinidos para ellos.

8. Subalgoritmos

Los *subalgoritmos*, *subprogramas* o *subrutinas* son trozos de código que pueden *reutilizarse* de forma *parametrizada*: las mismas sentencias se ejecutan sobre datos distintos o iguales. Cada utilización de un subprograma supone una *llamada* y los datos se pasan/reciben en forma de *argumentos* o *parámetros*.

La definición es única y se pone en la sección declarativa de un programa antes o después de la definición de tipos, constantes o variables. Los parámetros que recibe un subalgoritmo pueden ser:

a) De entrada (**E**): deben contener valores-R en la llamada que utilizará el subprograma. Dentro de un subalgoritmo, un parámetro formal de entrada no podrá modificarse (así no es necesario asumir que un parámetro de entrada se pasa por valor, lo cual es una cuestión de la implementación).
b) De salida (**S**): deben ser valores-L para contener información que calculará el subprograma.
c) De entrada-salida (**ES**): deben ser valores-L que además contendrán información de entrada al subprograma.

Por otro lado, podemos definir:

a) *Funciones*. Cada llamada representa un VALOR, todas las llamadas son (sub)expresiones (valores-R). La cabecera llevará un `<IdTipo>` detrás de la palabra `ALGORITMO`. El valor devuelto por una función se realiza con la sentencia **`DEVOLVER`** `<expresión>`, la cual también termina la ejecución de la función. Como norma de estilo, sólo debe aparecer una sentencia `DEVOLVER` en cada función. El tipo de la expresión `DEVOLVER` debe coincidir con el tipo de la función, compatible a su vez con el resto de la expresión donde aparece la llamada.

b) *Procedimientos*. Cada llamada representa una ACCIÓN, todas las llamadas a procedimientos son sentencias. La cabecera no llevará `<IdTipo>` detrás de la palabra `ALGORITMO`.

BNF	Definición en zona declarativa:
	`<Def_Subalgoritmo> ::=`
	`  **ALGORITMO** <CabeceraSubalgoritmo>`
	`  <DefinicionesLocales>`
	`  // **CONST**... **VAR**... **TIPOS**... <SubalgoritmosAnidados>`
	`  **INICIO**`
	`  <Sentencias>`
	`  **FIN** <IdSubalgoritmo>`
	`<CabeceraSubalgoritmo> ::=`
	`  [<IdTipo>] <IdSubalgoritmo>**(**<ListaParamFormales>**)**`
	`<ListaParamFormales> ::= <ParamFormal> {; <ParamFormal>}`
	`<ParamFormal> ::= (**ES**|**E**|**S**) <IdTipo> <id_variable> {, <id_variable>} |`
	`  (**ES**|**E**|**S**) <IdTipo> <id_array> {, <id_array>} |`
	`  (**ES**|**E**|**S**) <IdTipo> <id_registro> {, <id_registro>} |`
	`  (**ES**|**E**|**S**) <IdTipo> <id_puntero> {, <id_puntero>}`
	Llamadas en la zona ejecutable:
	`<Llamada> ::= IdSubalgoritmo**(** [<expr> {, <expr> }] **)**`

Ejemplo	`**ALGORITMO** **MiSubalgoritmo**`
	
	`  **ALGORITMO** ProcedimientoSuma(**E** N op1, op2; **S** resultado)`
	`  **INICIO**`
	`  resultado **=** op1+op2`
	`  **FIN** ProcedimientoSuma`
	
	`  **ALGORITMO** N FuncionSuma(**E** N op1, op2)`
	`  **VAR** N resultado`
	`  **INICIO**`
	`  resultado **=** op1+op2`
	`  **DEVOLVER** resultado`
	`  **FIN** FuncionSuma`
	
	`**VAR**  // no visibles en ProcedimientoSuma ni FuncionSuma`
	`  N suma1, suma2`
	`**INICIO**`
	`  ProcedimientoSuma(4, 3, suma1)`
	`  suma2 **=** FuncionSuma(4, 3)`
	`  Escribir(suma1, FuncionSuma(suma1, suma2))`
	`**FIN**`

Notas	Los tipos de los parámetros deben tener siempre nombre (tipos compuestos) o ser predefinidos.

Regla de ámbito: el ámbito de un identificador va desde el punto donde se define hasta el final del algoritmo que contiene la declaración.

Regla de ocultación: un identificador *oculta* a cualquier otro más externo con el mismo nombre.

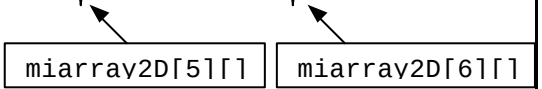
Una variable es *local* al declararla en un subalgoritmo y sólo puede utilizarse *"dentro de él"*.

Una variable local a un algoritmo se puede considerar como *global* a los subalgoritmos anidados.

Al llamar a un subalgoritmo se crean sus variables locales. Al terminar su ejecución, se destruyen.

9. Arrays

Los *arrays* son series de datos del mismo tipo, tipo *base*, que son accedidos con un mismo nombre, pero con un *índice* (valor de tipo ordinal) diferente para cada valor y consecutivos para valores adyacentes. Tienen una longitud que se fija en tiempo compilación, son *estáticos*.

<table>
<tr><td>BNF</td><td>
Declaración de un tipo array en la sección TIPOS:
<code><Def_TipoArray> ::= <IdTipoBase> <IdTipoArray> [<Rango>] {[<Rango>]}</code>
<code><Rango> ::= <ValorInicial>..<ValorFinal></code>
La sección VAR da la posibilidad de definir variables de tipo array sin nombre, o bien, de un tipo array ya definido con nombre:
<code><Def_Array> ::= <IdTipoBase> <id_array> [<Rango>] {[<Rango>]} |
<IdTipoArray> <id_array></code>
Acceso en la zona ejecutable a distintos niveles (según dimensiones):
<code><AccesoArray> ::= <id_array> { [<expr_ord>] }</code>
</td></tr>
<tr><td>Ejemplo</td><td>
<pre>
ALGORITMO TiposArrays
TIPOS
   N TArrayBidimensional[5..6][1..3]
VAR
   N numero
   N cuenta[1..10] = {10, 4, 5, 5, 4, 3, 7, 8, 7, 1}
   TArrayBidimensional miarray2D = { {1, 2, 3}, {0, 0, 0} }
INICIO
   numero = cuenta[5] // numero=4
   miarray2D[5][10] = 25
   Escribir(mearray2D) // <- error
FIN
</pre>


</td></tr>
<tr><td>Notas</td><td>
Se pueden declarar arrays en la zona de variables con tipos array sin nombre.
El tipo índice puede ser cualquier tipo ordinal. El tipo base cualquier tipo.
Las variables de tipo array se pueden inicializar con listas de valores entre { }. Incluso puede anidarse para arrays multidimensionales.
No se puede comparar arrays completos con ningún operador.
Los arrays pueden copiarse completos con una sola asignación si son del mismo tipo.
Una función puede devolver un array completo como valor.
No hay subprogramas que lean o escriban un array completo, exceptuando las cadenas de caracteres (arrays unidimensionales de tipo base <code>C</code>) que sí se pueden leer/escribir de forma completa.
</td></tr>
</table>

10. Cadenas de caracteres

Son arrays de tipo base `C` con algún valor añadido.

BNF	Definición de un tipo cadena en la sección **TIPOS**: `<Def_TipoCadena> ::= C <IdTipoCadena>[IndIni..IndFin]`

Ejemplo	**ALGORITMO** TiposCadenas
	TIPOS
	  C TCadena[1..80]
	CONST
	  TCadena cad_cte = “Esta no se puede modificar”
	VAR
	  TCadena micadena = “Esto es válido en C/C++”
	  C cadena[0..100] // también se pueden definir de tipo sin nombre
	INICIO
	  Leer(micadena) // <- valido
	  cadena = “¡Hola Mundo!”
	  ~~micadena = cadena~~ // <- error, tipos distintos
	  ~~cad_cte = cadena~~ // <- error, cad_cte no es L-Value
	  cadena[4] = ‘a’
	  Escribir(micadena)
	FIN

Notas	Se permite asignar literales de cadenas (entre comillas dobles) a un array de tipo cadena en su declaración o en la parte ejecutable. Se permite copiar cadenas con la asignación sólo si son del mismo tipo. Si son de longitud máxima distinta, se usará un subprograma diseñado para explícitamente ello. No se pueden comparar cadenas con los operadores relacionales, debe utilizarse un procedimiento diseñado explícitamente para ello. El terminador ‘`\0`’ o `CHR(0)` marca siempre el final lógico de una cadena (distinto del físico).

11. Registros

Los *registros* son variables compuestas de datos heterogéneos. Cada dato es un *campo*, que podrá ser de cualquier tipo, simple o compuesto. Se pueden anidar otros registros o arrays dentro de registros y un registro puede ser el tipo base de un array.

BNF	Definición de un tipo registro en la sección **TIPOS**:
	`<Def_TipoRegistro>::= REGISTRO <IdTipoRegistro>`
	`<Def_Campo> {<Def_Campo>}`
	`FINREGISTRO`
	`<Def_Campo> ::= <Def_VarSimple> | <Def_Array> | <Def_Registro>`
	Definición de un registro en la sección **VAR**:
	`<Def_Registro> ::= <IdTipoRegistro> <id_registro>`
	Acceso en la zona ejecutable:
	`<AccesoCampo> ::= <id_registro>.<id_campo>`

Ejemplo	`ALGORITMO TiposCadenas`
	`TIPOS`
	`  C TCadena[1..80]`
	`  REGISTRO TAlumno`
	`  N edad`
	`  TCadena nombre`
	`  FINREGISTRO`
	`VAR`
	`  TAlumno alumno1, alumno2`
	`INICIO`
	`  alumno1.nombre = “David”`
	`  alumno1.edad = 18`
	`  alumno2 = alumuno1`
	`  Escribir(alumno1)  // <- error`
	`FIN`

Notas	Se permite copiar registros completos con la asignación si son del mismo tipo. Se puede pasar un registro completo como parámetro de entrada a un subprograma. Una función puede devolver como valor un registro completo. No se puede leer ni escribir registros completos.

12. Ficheros

Desde el punto de vista de la programación imperativa, los ficheros son los únicos datos no volátiles, es decir, no se pierden entre las distintas ejecuciones de un mismo programa. Para trabajar con ellos, se define un tipo de datos especial, el tipo `FICHERO`. Las variables de este tipo son los *manejadores de fichero*, que contienen la información necesaria para trabajar con ellos desde programa, como por ejemplo la *posición* donde se va a efectuar la siguiente lectura o escritura. Para poder leer y/o escribir de/en un fichero es necesario *abrirlo*, momento en el que se asocia con un manejador de fichero. Cuando no se necesita trabajar más con el fichero, no necesariamente al finalizar el programa, éste se *cierra* y queda desasociado el manejador del fichero.

Distinguiremos dos tipos de fichero, los de texto, que contienen la información codificada en caracteres del conjunto estándar de la máquina, y los binarios, que contienen la información en el mismo formato en que están en la memoria principal cuando se está ejecutando el programa. En un programa la única diferencia entre ambos será la forma de leer y/o escribir los datos de/en fichero. Para los ficheros de texto se utilizan las subrutinas estándares de lectura/escritura formateada, para los binarios se utilizan las de lectura/escritura no formateada. Las primeras realizan la conversión necesaria del formato de memoria principal al conjunto de caracteres de la máquina, las últimas no realizan tal conversión. El anexo III describe las especificaciones de todas las subrutinas estándares de gestión de ficheros de nuestro pseudolenguaje.

BNF	Definición de un manejador de fichero en la sección **VAR**: `<Def_Fichero>::= FICHERO <id_manejador>` En la zona ejecutable se utilizará como parámetro para las subrutinas de gestión de ficheros.

Ejemplo	**ALGORITMO CopiarFicheroTexto**
	VAR
	  FICHERO ffuente, fdestino
	  Z numero
	INICIO
	  ffuente = Abrir(“NombreViejo.txt”)
	  fdestino = Abrir(“NombreNuevo.txt”)
	  Leer(ffuente, numero) // lectura previa
	  MIENTRAS NO Fin(ffuente) **HACER**
	  Escribir(fdestino, numero)
	  Leer(ffuente, numero)
	  FINMIENTRAS
	  Cerrar(ffuente)
	  Cerrar(fdestino)
	FIN

Notas	Las escrituras sobre ficheros son permanentes, de tal forma que los cambios realizados por un subprograma afectan al resto.

13. Punteros

Los punteros se consideran variables de tipo simple a través de los cuales se puede acceder a otras variables sin nombre, las variables *anónimas* de un tipo base `<TBase>`, que están ubicadas en la zona de memoria dinámica gestionada directamente por el programador. El tipo base de la variable anónima o apuntada puede ser cualquiera, simple o compuesto.

Con `ASIGNAR(<puntero>)` se reserva memoria para la variable anónima en memoria dinámica. Con `LIBERAR(<puntero>)` se libera la memoria a la que apunta el puntero para que pueda ser reutilizada. El valor del puntero queda indefinido tras la liberación.

`NULO` es un valor constante compatible con cualquier tipo puntero. Indica que el puntero no apunta a nada válido. Un puntero se inicializa con `NULO`, con el valor de otro puntero o con `ASIGNAR()`.

Los únicos operadores primitivos que pueden utilizarse con punteros son los de igualdad (==) y desigualdad (!=). También pueden asignarse uno a otro, pasarse como parámetro a un subprograma y devolverlo una función.

BNF

Declaración de tipos puntero en la sección **`TIPOS`**:

```
<Def_TipoPuntero> ::= <TipoBase> *<TipoPuntero>
<Def_TipoArrayP> ::= <TipoBase> *<TipoArrayP> [<Rango>]
```

En la sección **`VAR`** se pueden definir punteros de tipo sin nombre o punteros de un tipo puntero ya definido previamente:

```
<Def_Puntero> ::= <TipoPuntero> <id_puntero> |
                  <TipoBase> *<id_puntero>

<Def_ArrayP>  ::= <TipoArrayP> <id_array_p> |
                  <TipoBase> *<id_array_p> [<Rango>]
```

Acceso a la variable anónima o apuntada:

```
<AccesoAnónimo> ::= *<id_puntero>
<AccesoCampo> ::=<id_puntero>-><id_campo> | (*<id_puntero>).<id_campo>
<AccesoElemento> ::= *<id_puntero>[<expr_ord>]
```

Ejemplo

```
ALGORITMO TiposPunteros
TIPOS
   N *TPunteroN
   N *TArrayPunterosN[1..10]
   TNodo *TPunteroNodo
   REGISTRO TNodo
      N valor
      TPunteroNodo sig // <- autorreferencia
   FINREGISTRO
VAR
   TPunteroN mipunteroN
   TArrayPunterosN miarrayPN
   TPunteroNodo lista
INICIO
   ASIGNAR(mipunteroN)
   *mipunteroN = 5
   ASIGNAR(lista)
   (*lista).valor = 5
   lista->sig = NULO
   miarrayPN[1] = mipunteroN
   LIBERAR(mipunteroN)
   LIBERAR(lista)
FIN
```

Notas	Una estructura de datos *autorreferenciada* (registro o clase) contiene punteros a sí misma. Para poder definirlas se permite definir previamente el tipo puntero a la estructura de datos. La propia estructura debe definirse inmediatamente después. (No obstante, C++ permite utilizar el tipo puntero a la estructura sin que se defina previamente).

14. Clases

Una *clase* es una entidad que encapsula datos, que llamaremos *atributos*, y subprogramas miembros, que llamaremos *métodos*, que manipulan esos datos. Distinguiremos entre datos y métodos *privados*, ocultos desde el exterior, y datos y métodos *públicos*, accesibles desde el exterior. Los atributos podrán ser tipos de datos definidos por el usuario, o datos propiamente dichos, como variables o constantes.

Siguiendo el principio de ocultación de la información, cualquier manipulación sobre los datos privados, deberá hacerse forzosamente a través de llamadas a los métodos públicos, llamadas que denominaremos *mensajes*.

Un *objeto* es una *instanciación* de una clase o, en otras palabras, una variable de un tipo clase. La instanciación de un objeto supone su creación completa al ejecutarse su *constructor*. Su llamada se realiza implícitamente al definirse el objeto en la sección VAR, definición que podrá contener argumentos de inicialización entre paréntesis. Su destrucción se realiza ejecutándose su *destructor*. Su llamada se realiza de forma implícita al terminar la ejecución del algoritmo que lo define (igual que cualquier varible local). El constructor y el destructor deben definirse obligatoriamente como métodos de la clase con los nombres `Crear()` y `Destruir()`, aunque estén vacíos. Deberá haber siempre un destructor sin argumentos y, como mínimo, un constructor sin argumentos. Si se define más de un constructor, se distinguirán por los argumentos que reciban, por ejemplo, un *constructor de copia* siempre recibirá como parámetro un objeto, y sólo uno, de la misma clase.

Se puede asignar un objeto a otro siempre que sean de la misma clase. Si hay constructor de copia, se llama primero al destructor para el objeto destino, y después al constructor de copia de la clase para asignar el objeto fuente al destino. Si no hay definido un constructor de copia, simplemente se realizará la asignación primitiva, con la consiguiente incoherencia cuando haya datos en memoria dinámica.

Se pueden definir constantes y tipos públicos dentro de la interfaz de una clase. Se accederá a ellos desde el exterior con el *operador de alcance* `::` (al estilo de C++): `<ClasePropietaria>::<Tipo>` y `<ClasePropietaria>::<Constante>`

También pueden definirse *punteros a objetos*. En este caso, el objeto se instancia, llamando implícitamente al constructor, al asignar memoria en la sección ejecutable con la primitiva `ASIGNAR(<p_objeto>)`. Si el constructor espera argumentos, éstos se pondrán entre paréntesis: `ASIGNAR(<p_objeto>(<argumentos>))`. El objeto podrá destruirse en la sección ejecutable mediante una llamada a la primitiva `LIBERAR(<p_objeto>)`.

Un objeto puede ser un atributo de una clase, dando lugar a la *composición de clases*. Cuando se crea/destruye una clase que contiene otras clases como atributos, se llama automáticamente a su constructor/destructor así como a los constructores/destructores de las clases anidadas.

Nuestro pseudolenguaje no contempla otras características típicas de la orientación a objetos como la *herencia* ni el *polimorfismo*, ya que está pensado para alumnos noveles en la programación. Tampoco es orientado al objeto puro, ya que podrán mezclarse los objetos con algoritmos externos que hacen uso de los primeros.

Todos los métodos y atributos, tanto públicos como privados, tienen ámbito de clase: pueden ser accedidos desde cualquier método de la clase, es decir, cualquier objeto de esa clase tiene acceso libre a ellos. Desde algoritmos externos, sólo los públicos pueden ser accedidos a través de un objeto instanciado o un puntero a un objeto ya creado.

Cada objeto puede referenciarse mediante el puntero a sí mismo definido implícitamente como:

```
<MiClase> *este
```

útil cuando parámetros o variables locales de un método se llaman igual que los atributos. Ver método `Dimensionar()` del ejemplo `<CRectangulo>` más adelante.

Distinguiremos dos partes en la definición de una clase, cada una de las cuales contendrá:

a) *Interfaz*, tipos públicos, constantes públicas, atributos públicos y cabeceras de los métodos públicos.

b) *Implementación*, tipos privados, constantes privadas y atributos privados[4] y cuerpos de los métodos públicos y privados.

BNF

Definición de clases:

```
<Def_Clase> ::=

   INTERFAZ CLASE <Id_Clase>   // parte visible
      TIPOS
         <Def_tipo> { <Def_tipo> }
      ATRIBUTOS
         <Def_cte_o_var> { <Def_cte_o_var> }
      METODOS // interfaces visibles
         <CabeceraSubalgoritmo> { <CabeceraSubalgoritmo> }
   FIN <Id_Clase>

   IMPLEMENTACION CLASE <Id_Clase>  // parte oculta
      TIPOS
         <Def_tipo> { <Def_tipo> }
      ATRIBUTOS
         <Def_cte_o_var> { <Def_cte_o_var> }
      METODOS // visibles y ocultos
         <Def_Subalgoritmo> { <Def_Subalgoritmo> }
   FIN <Id_Clase>
```

```
<Def_cte_o_var> ::=  <Def_ConstSimb> | <Def_VarSimple> |
                     <Def_Array> | <Def_Registro> | <Def_Objeto>
```

La definición de objetos o punteros a objetos se hace en la sección **VAR**:

```
<Def_Objeto> ::= <Id_Clase> <id_objeto>[(<lista_inicializadores>)]
```

```
<DeclPunteroObj> ::= <Id_Clase> *<id_puntero_obj>
```

Los mensajes se basan en las llamadas convencionales a subprogramas y se realizan en la sección ejecutable. Pueden realizarse con los objetos o a través de punteros a objetos:

```
<Mensaje> ::=  <id_objeto>.<Método> |
               (*<id_puntero_obj>).<Método> |
               <id_puntero_obj>-><Método>
```

La misma notación se utiliza para acceder a atributos públicos:

```
<Acceso_atrib> ::= <id_objeto>.<id_atributo> |
               (*<id_puntero_obj>).<id_atributo> |
               <id_objetoero_obj>-><id_atributo>
```

El acceso a tipos públicos y constantes públicas se realiza con el operador de alcance '**::**':

```
<Acceso_clase> ::= <IdClase>::<id_cte_o_tipo>
```

[4] En el lenguaje C++, todos los atributos, públicos y privados, se definen junto a las cabeceras de los métodos, públicos y privados. No obstante, se distinguen por las palabras reservadas `public` y `private`.

Ejemplo

```
INTERFAZ CLASE CRectangulo
METODOS
   Crear()
   Destruir()
   Dimensionar(E N nuevo_ancho; E N nuevo_largo)
   R Area()
   R Perimetro()
FIN CRectangulo

IMPLEMENTACION CLASE CRectangulo
ATRIBUTOS
   R ancho, largo

Crear()
INICIO
   ancho = 0;   largo = 0
FIN

Destruir()
INICIO
FIN

Dimensionar(E N ancho; E N largo)
INICIO
// aquí este->... son los atributos y ancho/largo son los parámetros
   este->ancho = ancho;   este->largo = largo
FIN

R Area()
INICIO
   DEVOLVER ancho * largo
FIN

R Perimetro()
INICIO
   DEVOLVER ancho*2.0 + largo*2.0
FIN

FIN CRectangulo


ALGORITMO Principal
CONST
   AnchoA4 = 210;   LargoA4 = 297   // mm
VAR
   CRectangulo a4 // creación implícita por defecto
INICIO
   a4.Dimensionar(AnchoA4, LargoA4)
   Escribir("El área de una lámina A4 es: ", a4.Area() )
   // destrucción implícita
FIN
```

15. Sentencias

Asignación	<asignación> ::= <valor_L> **=** <expresión> <valor_L> ::= <id_variable> | <ref_var_apuntada> | <ref_campo_o_reg> | <ref_array_o_componente>
Selección	<selección> ::= **SI** <expr_B> **ENTONCES** {<sentencia>} { **SINOSI** <expr_B> {<sentencia>} } [**SINO** {<sentencia>}] **FINSI**
Selección	<SelecciónMúltiple> ::= **CASO** <expr_ord> **SEA** <ListaValores>: {<sentencia>} { <ListaValores>: {<sentencia>} } [**SINO** {<sentencia>}] **FINCASO** <ListaValores> ::= <Val> {, <Val> } <Val> ::= <Valor_ord> | <Rango>
Iteración	<bucle_mientras> ::= **MIENTRAS** <expr_B> **HACER** {<sentencia>} **FINMIENTRAS** <bucle_repetir>::= **REPETIR** {<sentencia>} **HASTA QUE** <expr_B> <bucle_para>::= **PARA** <id_ord>**=**<exp_ord> **HASTA** <expr_ord>[**PASO** <expr_Z>] **HACER** {<sentencia>} **FINPARA**
Llamada a procedimiento[5]	<Llamada_Proc> ::= <Id_Proc>**(**<arg> {, <arg>} **)**

[5] Una llamada a una función no es una sentencia, es una (sub)expresión.

16. Notas sobre estilo

- Los tipos llevarán una 'T' como primera letra y cada letra de principio de palabra en Mayúsculas. Ejemplos: `TCadena`, `TMiTipo`, `TMiArrayBidimensional`.
- Las clases empezarán con la 'C' como primera letra y cada letra de principio de palabra en Mayúsculas. Ejemplos: `CCubo`, `CMiCoche`, `CJuego`.
- Los nombres de (sub)algoritmos tendrán en mayúscula cada letra de principio de palabra y preferentemente serán o contendrán un verbo. Ejemplos: `Sumar`, `EsLetra`, `Convertir`.
- Las variables irán en minúsculas y tendrán un identificador representativo de su posible valor. Ejemplos: `mi_cadena`, `numero`, `carta`.
- Debe incluirse comentarios para clarificar el significado del código, especiamente como cabecera de todos los (sub)algoritmos.
- Se debe realizar una indentación en instrucciones que estén anidadas en otras, de al menos 2 espacios y no superior a 4 espacios.
- Se utilizarán líneas en blanco y se partirán las largas en puntos que clarifiquen su lectura.
- No se debe utilizar un parámetro de salida (**S**) salvo para almacenar información[6].

El siguiente programa cumple las normas de estilo:

```
ALGORITMO MayorMenor
/* calcula el mayor y el menor de tres números */
VAR
   N mayor, menor, num1, num2, num3

INICIO
   /* entrada */
   Escribir("Introduce 3 numeros: ")
   Leer(num1)
   Leer(num2)
   Leer(num3)

   /* procesamiento */
   SI num1<num2 ENTONCES
      menor = num1
      mayor = num2
   SINO
      menor = num2
      mayor = num1
   FINSI
   SI num3<menor ENTONCES
      menor = num3
   SINO
      SI num3>mayor ENTONCES
         mayor = num3
      FINSI
   FINSI

   /* salida */
   Escribir(“El mayor es: ”, mayor)
   Escribir(“El menor es: ”, menor)

FIN.
```

[6] Recuérdese que no se permite modificar un parámetro de entrada E.

Anexo I. Operadores

En pseudolenguaje los **tipos simples** disponen de los siguientes dominios y operadores:

Tipo	*Dominio*	*Operadores binarios*	*Operadores unarios*
N	Números naturales	`== != < <= > >= + - * DIV MOD`	(ninguno)
Z	Números enteros	`== != < <= > >= + - * DIV MOD`	`+ -`
R	Números reales	`== != < <= > >= + - * /`	`+ -`
C	Conjunto caracteres de e/s	`== != < <= > >=`	(ninguno)
B	`VERDADERO, FALSO`	`== != < <= > >= Y O`	`NO`
enumerado	Definido por el programador	`== != < <= > >=`	(ninguno)
puntero	`NULO`, direcciones físicas	`== !=`	(ninguno)

Con las siguientes *reglas de precedencia*[7]:

1º. Paréntesis	`( )`
2º. Operadores de accesibilidad	`. [] * ::`
3º. Operadores unarios	`NO - +`
4º. Operadores multiplicativos	`* / DIV MOD`
5º. Operadores aditivos	`+ -`
6º. Operadores relacionales	`< > <= >=`
7º. Operadores de igualdad	`== !=`
8º. Operador lógico de conjunción	`Y`
9º. Operador lógico de disyunción	`O`

Los dominios de los **tipos compuestos** dependen de los dominios de los tipos componentes. Y no se puede aplicar ningún operador a las variables completas de tipo compuesto, ni siquiera a cadenas de caracteres: se deben definir los subprogramas apropiados para implementar cualquier operación.

No obstante, las variables de tipo compuesto sí se pueden asignar cuando el tipo sea exactamente el mismo. Igualmente, se permite pasar tipos compuestos como parámetros a subprogramas y devolverlos como valor de una función.

La asignación primitiva de **estructuras de datos basadas en punteros**, no involucra la asignación de las variables anónimas apuntadas por cualquier puntero. La creación, destrucción o cualquier otra operación de estas estructuras debe ser implementada por el programador, no son operaciones predefinidas en pseudolenguaje.

[7] El operador * de accesibilidad se usa con punteros y no tiene nada que ver con el operador aritmético * multiplicativo.

Anexo II. Subprogramas predefinidos

Podemos considerar predefinidos los siguientes subprogramas:

Matemáticas

```
Z ABS(<expr_numérica>)
R SQRT(<expr_numérica_R>)
Z POW(<expr_numérica_Z>, <expr_numérica_Z>)
R POW(<expr_numérica_R>, <expr_numérica_R>)
```

Misceláneas

```
C CAP(<carácter>)
C CHR(<expr_N>)
N ORD(<expr_ordinal>)
<TipoOrdinal> PRED(<expr_ordinal>)
<TipoOrdinal> SUCC(<expr_ordinal>)
N TAMAÑO(<Tipo>)
ASIGNAR(<puntero>)
LIBERAR(<puntero>)
```

Memoria

```
B DISPONIBLE(<puntero>)
ASIGNAR(<puntero>)
ASIGNAR(<p_objeto>(<argumentos>))
LIBERAR(<puntero>)
```

Anexo III. Gestión de ficheros

```
FICHERO Abrir(E <TCadena> <nombre_fichero>)
FICHERO Crear(E <TCadena> <nombre_fichero>)
B       Existe(E <TCadena> <nombre>)
        Cerrar(E FICHERO <manejador>)
```

`Abrir` y `Crear` abren un fichero con el nombre especificado. El nombre es el que utiliza el sistema operativo y puede contener el camino completo en la estructura de archivos. Si no existe el fichero, `Abrir` no lo crea. Si existe el fichero, `Crear` lo borra antes de abrirlo. `Existe` comprueba la existencia del fichero o directorio especificado en el sistema de archivos. `Cerrar` cierra un fichero.

```
        Leer(E FICHERO <manejador>; E <TSimple_o_TCadena> <variable>
                                  {; E <TSimple_o_TCadena> <variable>})
        Escribir(ES FICHERO <manejador>; E <TSimple_o_TCadena> <expresión>
                                       {; E <TSimple_o_TCadena> <variable>})
```

Lectura y escritura formateada. El tamaño de los objetos se calcula automáticamente por el tipo de cada variable o expresión. `Leer` pone el dato leído en las variables especificadas y `Escribir` escribe el resultado de las expresiones en el fichero. Ambos subprogramas avanzan automáticamente la posición para la siguiente operación de lectura/escritura en el número de bytes adecuado.

```
        LeerBin(E FICHERO <manejador>; E <TCualquiera> <var>; E N <nbytes>)
        EscribirBin(ES FICHERO <manejador>; E <TCualquiera> <var>;E N <nbytes>)
```

Lectura y escritura no formateada. `Leer` lee `<nbytes>` del fichero y los pone en `<var>`. `Escribir` escribe `<nbytes>` de la `<var>` especificada. Ambos subprogramas avanzan automáticamente la posición para la siguiente operación de lectura/escritura en el número de bytes especificado.

```
B       Fin(E FICHERO <manejador>)
N       Posicion(E FICHERO <manejador>)
N       Longitud(E FICHERO <manejador>)
        Eliminar(E FICHERO <manejador>; E N <nbytes>)
        Situar(E FICHERO <manejador>; E N <posición>)
```

`Fin` devuelve VERDADERO al llegar al final de fichero, FALSO si no. `Posición` devuelve el nº de byte donde se efectuará la siguiente operación de lectura/escritura. La posición 0 es la primera. `Longitud` devuelve el tamaño del fichero en bytes. `Situar` establece la posición (número de byte) donde se va a realizar la siguiente operación de lectura/escritura. `Eliminar` elimina el número de bytes especificado a partir de la posición actual.

```
        Renombrar(E <TCadena> <nombre_antiguo>, <nombre_nuevo>)
        Borrar(E <TCadena> <nombre_fichero>)
        CrearDir(E <TCadena> <nombre_dir>)
        BorrarDir(E <TCadena> <nombre_dir>)
```

`Renombrar` renombra un fichero o directorio del sistema de archivos que debe existir. `Borrar` borra un fichero. `CrearDir` crea un nuevo directorio en el sistema de archivos y `BorrarDir` borra un directorio.