PROGRAMACIÓN II (SISTEMAS) - CURSO 2002/03

EIFFEL ESTRUCTURADO (I)

1. INTRODUCCIÓN

1. 1. Características de Eiffel

Eiffel es un lenguaje de programación creado por Bertrand Meyer en 1988. Entre las características que le hacen adecuado para la asignatura Programación II destacan el ser un lenguaje estructurado, con comprobación estricta de tipos, soporte para programación bajo contrato y verificación de programas. La característica fundamental de ser orientado a objeto será utilizada en cursos posteriores. Al no contemplarla en la asignatura, la descripción que se va a proporcionar a continuación será **muy limitada** y no contemplará partes esenciales del lenguaje.

- Un programa en Eiffel consiste en un conjunto de **clases**, cada una de las cuales está formada por **atributos** (datos) y **métodos** (código, equivalente a subprogramas). Cada clase se almacena en un fichero con extensión `.e` y en el momento de compilar el programa se indica la clase y el método de ella que sirve como punto de entrada al programa.
- Asociaremos el concepto de módulo al de método de una clase, y por lo tanto un método será el equivalente a un subprograma en Pascal. Sin embargo, es necesario tener en cuenta que esta asociación *no es correcta* desde el punto de vista de la programación orientada a objetos, y que por lo tanto el alumno deberá cambiar de punto de vista en cursos posteriores.
- En Eiffel (como en la mayoría de lenguajes orientados a objeto) la definición de nuevos tipos de datos por el programador se sustituye por la definición de clases, que es un concepto más potente (una clase define no sólo la manera de representar los datos sino además las operaciones que se pueden efectuar sobre ellos).
- Por lo tanto, cualquier tipo de datos que no sea simple[1] se representa mediante una clase en Eiffel (esto incluye a los tipos de datos array, registro, cadena de caracteres, etc.). Los valores de estos tipos de datos se denominan **objetos** y se deben crear en tiempo de ejecución. Las variables cuyo tipo de datos es una clase almacenan **referencias** a objetos.

[1] En realidad, también los tipos de datos simples son clases, predefinidas en el lenguaje. Sin embargo, tienen características especiales que hacen que se puedan usar de la misma forma que los tipos de datos de Pascal.

- Trabajar con objetos se realiza de una forma parecida (salvando las distancias) a la manera en que se trabaja con variables dinámicas en Pascal.
- Cualquier operación que afecte a un objeto (acceso a un elemento de un array, por ejemplo) se realiza llamando[2] al método adecuado de entre los que se hallan definidos en la clase correspondiente. Esta llamada tiene una sintaxis especial.

1. 2. Descripción de la sintaxis

Para describir la sintaxis del lenguaje se utilizará una versión simplificada de las formas normales de Backus-Naur. Cada concepto se definirá en una o más líneas con el formato *nombre* ::= *definición*. En la parte de la definición, lo que aparezca en **negrita** se debe escribir literalmente, y lo que aparezca en *cursiva* hace referencia al nombre de un concepto definido anteriormente o bien es un nombre descriptivo de lo que debe aparecer en esa parte. Además, los siguientes símbolos tienen un significado especial:

- Lo que aparezca encerrado entre corchetes [ ] representa una parte opcional, que puede o no aparecer.
- Lo que aparezca encerrado entre llaves { } representa una parte que se puede repetir cero o más veces.
- La construcción *a* | *b* se utiliza para indicar una alternativa: O bien aparece *a* o bien *b*.

Cuando los símbolos anteriores aparezcan encerrados en una caja [ ] { } se interpretará que se deben escribir literalmente. Definiciones básicas:

comentario ::= - - *texto_descriptivo*

tipo_simple ::= BOOLEAN | CHARACTER | INTEGER | DOUBLE

declaración_vars ::= { *lista_vars* **:** *tipo* [**;**] }
lista_vars ::= *nombre_var* { **,** *nombre_var* }

clase ::= [**indexing** "*descripción*"] **class** *nombre* **creation** *lista_métodos* **feature** { *atributo* | *método* } **end**
lista_métodos ::= *nombre_método* { **,** *nombre_método* }
atributo ::= *declaración_vars*

1. 3. Compilación y ejecución

En las prácticas de la asignatura se utilizara el compilador **SmallEiffel** (recientemente a cambiado su nombre a **SmartEiffel**), instalado en duero, jair, las estaciones de trabajo SUN y los PCs en el entorno Linux. La versión utilizada es la -0.74, el compilador y las librerías estándar son software libre con licencia GNU, y se puede descargar para uso personal desde su página oficial, http://www.loria.fr/projets/SmallEiffel, o bien desde el ftp del departamento. Existen versiones para el entorno Windows.

Los programas Eiffel son ficheros de texto. Se recomienda utilizar el editor **gvim**, y activar la opción de resaltado de sintaxis.

[2] En terminología de programación orientada o objetos este proceso (llamar a un método) se denomina *enviar un mensaje* al objeto correspondiente.

Escriba el siguiente programa utilizando **gvim** y guárdelo con el nombre **ejemplo.e**:

```
indexing "Programa de ejemplo en Eiffel"
class EJEMPLO
creation primero,segundo
feature
  primero is
  do
    io.put_string("Primer punto de entrada%N");
  end -- primero
  segundo is
  do
    io.put_string("Segundo punto de entrada%N");
  end -- segundo
end -- EJEMPLO
```

Este programa consta de una clase con dos métodos que escriben una cadena de texto por la salida estándar (generalmente la pantalla). Cualquiera de estos métodos puede servir de punto de entrada al programa. Al compilar el programa se debe indicar el nombre del fichero que define la clase y el nombre del método que sirve como punto de entrada (cualquiera de los métodos listados en el apartado **creation**). Salve el programa a disco, salga del editor y teclee:

- `compile ejemplo primero`
- `clean ejemplo`

El primer comando compila el programa indicando que su punto de entrada es el método **primero**. Si no se muestra ningún mensaje en pantalla significa que no se han producido errores y se ha generado un fichero ejecutable con el nombre **a.out**. El segundo comando borra los ficheros auxiliares generados en la compilación (**Nota**: Estos ficheros no son necesarios y pueden llegar a ocupar mucho espacio en disco, por lo que es muy importante ejecutar este comando siempre que se compile un programa). Si ahora escribe:

- `a.out`

Se ejecuta el programa (en este caso deberá aparecer el mensaje "Primer punto de entrada" en pantalla). Si teclea:

- `compile ejemplo segundo`
- `clean ejemplo`
- `a.out`

La ejecución del programa mostrará "Segundo punto de entrada" en pantalla.

2. MODULARIDAD

2. 1. Definición de métodos

método ::= *nombre* [**(** *declaración_vars* **)**] [**:** *tipo_resultado*] **is**
[**local** *declaración_vars*]
[**require** *asertos*] **do** *sentencias* [**ensure** *asertos*] [**rescue** *sentencias*] **end**

asertos ::= *cláusula* { **;** *cláusula* }

cláusula ::= [*etiqueta_descriptiva* **:**] *expresión_lógica* | *comentario*

Los métodos (*subprogramas*) de Eiffel constan de una cabecera, una zona opcional de declaración de variables locales y una secuencia de sentencias que realizan la tarea del módulo. Existen dos zonas opcionales donde se escriben las precondiciones (**require**) y poscondiciones (**ensure**) del método. Además, es posible definir una zona (**rescue**) que realice el tratamiento de errores que se hallan producido al ejecutar el método. En la cabecera se define el nombre del método, los parámetros de entrada y el valor devuelto (si es una función). El esquema es parecido al de Pascal, pero con las siguientes diferencias esenciales:

- No existen palabras reservadas en Eiffel que distingan entre procedimientos y funciones (como **procedure** y **function** en Pascal). Si en la cabecera aparece un tipo resultado, entonces es una función y si no aparece es un procedimiento.
- A diferencia de Pascal, no es posible definir subprogramas dentro de otros subprogramas.
- Todos los parámetros se pasan por valor. No existe el paso de parámetros **por variable**. Esto no supone una pérdida de funcionalidad cuando los parámetros son objetos (equivalentes a datos estructurados en Pascal) debido a que siempre es posible cambiar el estado de un objeto (es decir, alguno de sus valores almacenados) mediante llamadas a alguno de sus métodos. Si lo que se desea es devolver un valor de tipo simple, se define una función que devuelva un valor de ese tipo.
- El problema puede surgir cuando se desea devolver más de un valor de tipo simple (por ejemplo, un método que indique el valor máximo y mínimo de un array de enteros). Para estos casos se recomienda crear varios métodos, cada uno de los cuales devuelva un único valor (en el ejemplo anterior, crear un método que devuelva el mínimo y otro que devuelva el máximo). En cursos posteriores se verán otras técnicas que se pueden emplear para este caso.
- Las funciones en Eiffel devuelven el valor asignándolo a la variable predefinida **Result**. El enfoque es exactamente el mismo que en Pascal, salvo que la variable resultado en lugar de tener el nombre de la función tiene un nombre genérico.
- Las variables asociadas a los parámetros son de sólo lectura: No se les puede asignar valores.

2. 2. Llamada a un método

llamada_a_método_local ::= *nombre_método* [**(** *parámetros_actuales* **)**]

llamada_a_método_objeto ::= *referencia_objeto* **.** *nombre_método* [**(** *parámetros_actuales* **)**]

La llamada a un método perteneciente a la clase que estamos creando se realiza exactamente igual que en Pascal. Si es un procedimiento, la llamada es una sentencia por si misma, y si es una función la llamada debe encontrarse dentro de una expresión. Lo único que se debe tener en cuenta es que todos los parámetros son por valor.

Si queremos llamar a un método definido en otra clase la sintaxis es distinta, se debe escribir una **referencia**[3] (o un valor, en el caso de tipos simples) a un objeto cuyo tipo de datos sea la clase donde está definido el método, un punto, y la llamada al método. El significado de ésta construcción es realizar la operación indicada sobre el objeto.

Si comparamos la forma de calcular el máximo de dos valores utilizando un método local y el método predefinido en la clase INTEGER, se puede observar que en segundo caso la función máximo sólo tiene un parámetro, no dos: Esto no es casualidad, en éste segundo caso se está pidiendo a un objeto de tipo entero que calcule el máximo no de dos enteros cualquiera, sino de *él mismo* y el otro entero que se pasa por parámetro.

```
max(a,b: INTEGER): INTEGER is
do
  if a > b then
    Result := a
  else
    Result := b
  end
end -- max

test is
local x,y,z: INTEGER
do
  x := 3; y := 5;
  z := max(x,y);
  z := max(y,x);
  z := max(x+2,y-2);
  z := max(7,8);
end -- test
```

```
-- max es un método definido
-- en la clase INTEGER

test is
local x,y,z: INTEGER
do
  x := 3; y := 5;
  z := x.max(y);
  z := y.max(x);
  z := (x+2).max(y-2);
  z := (7).max(8);
end -- test
```

En la siguiente sección se muestran algunos métodos pertenecientes a clases predefinidas en Eiffel. En todos ellos se puede apreciar, respecto a su versión equivalente en Pascal, que falta un parámetro, que implícitamente es el objeto que sirve para seleccionar el método utilizado.

2. 3. Métodos útiles

A continuación se describen algunos métodos que pueden ser de utilidad en la creación de programas en Eiffel. En Pascal, estos métodos se corresponderían a funciones y procedimientos predefinidos del lenguaje. Como Eiffel está orientado a objetos, todos las operaciones deben estar *definidas* en las clases adecuadas y deben llamarse utilizando objetos de esa clase.

Las operaciones de tipo matemático suelen estar asociadas a los tipos de datos (clases) INTEGER o REAL. Aquellas que tratan con caracteres al tipo CHARACTER, y las operaciones de entrada/salida estándar están asociadas a clases que representan ficheros de texto.

Las operaciones de lectura por teclado y escritura en pantalla pertenecen a ésta última categoría. En el entorno SmallEiffel, cualquier clase incluye por defecto una variable, **io**, que almacena una referencia a un objeto cuya clase es STD_INPUT_OUTPUT (fichero de texto de entrada/salida) y que

[3] El concepto de *referencia* se verá con un poco más de detalle en el apartado dedicado a arrays y cadenas de caracteres. De momento se puede asociar (salvando las distancias) el concepto de *referencia* al de *puntero* a un objeto.

está asociado (el objeto) al teclado (entrada estándar) y a la pantalla (salida estándar). Las operaciones de lectura y escritura en éste fichero corresponden a lecturas del teclado y escrituras en la pantalla, respectivamente.

En lugar de describir directamente la cabecera de los métodos, se ha optado por mostrar un ejemplo de su uso, para que sea más sencilla su utilización. Las variables cuyo nombre comienza por *b* son de tipo BOOLEAN, las que comienzan por *c* son de tipo CHARACTER, por *i* de tipo INTEGER, por *d* de tipo DOUBLE y por *s* de tipo STRING:

Salida por pantalla	io.put_boolean(b)	Estos métodos traducen el valor a su representación textual y la muestran por pantalla.
	io.put_character(c)	
	io.put_integer(i)	
	io.put_double(d)	
	io.put_string(s)	
	io.put_new_line	
Lectura por teclado	io.read_character c := io.last_character	La lectura de un dato se realiza en dos pasos: Primero se pide leer un dato del tipo adecuado y luego se obtiene el valor leído.
	io.read_integer i := io.last_integer	
	io.read_double d := io.last_double	
	io.read_line s := io.last_string	
Conversión entre tipos	i := c.code	Código ASCII del carácter almacenado en *c*.
	c := i.to_character	Carácter cuyo código ASCII es el valor almacenado en *i*.
	c2 := c1.to_upper	Conversión a mayúsculas
	c2 := c1.to_lower	Conversión a minúsculas
	if c1.same_as(c2) then …	Cierto si los caracteres almacenados en *c1* y *c2* son iguales sin diferenciar entre mayúsculas y minúsculas
	s := i.to_string	Traduce el valor almacenado en *i* a una cadena de texto. En el segundo caso la cadena tendrá 8 caracteres como mínimo y se rellenará con espacios a la izquierda si es necesario.
	s := i.to_string_format(8)	
	s := d.to_string	Traduce el valor almacenado en *r* a una cadena de texto. En el primer caso se escriben todos los decimales y en el segundo caso se escribirán 6 decimales.
	s := d.to_string_format(6)	
	i := d.floor	Devuelve el mayor entero menor o igual que el valor de *d*
	i := d.ceiling	Devuelve el menor entero mayor o igual que el valor de *d*
	i := d.rounded	Redondeo al valor entero más cercano
Matemáticos y otros	i2 := i1.abs	Devuelve el valor absoluto
	d2 := d1.abs	
	i3 := i1.min(i2)	Devuelve el mínimo de los valores almacenados en *i1* e *i2*
	i3 := i1.max(i2)	Devuelve el máximo de los valores almacenados en *i1* e *i2*
	d3 := d1.pow(d2)	Devuelve la potencia del valor de *d1* elevado al valor de *d2*
	d2 := d1.sqrt	Devuelve la raíz cuadrada del valor almacenado en *d1*
	d2 := d1.exp	Devuelve la exponencial del valor almacenado en *d1*
	d2 := d1.log	Devuelve el logaritmo neperiano del valor almacenado en *d1*
	d2 := d1.log10	Devuelve el logaritmo decimal del valor almacenado en *d1*
	c2 := c1.previous	Anterior carácter en la tabla de códigos
	c2 := c1.next	Siguiente carácter en la tabla de códigos

3. VARIABLES

3. 1. Tipos simples de datos

Los tipos simples de Eiffel sirven para representar valores como números o caracteres que no pueden dividirse en partes más sencillas. Se utilizan de manera muy parecida a sus equivalentes en Pascal, aunque en realidad son un tipo de clases que recibe un tratamiento especial para facilitar su uso. En la tabla siguiente se muestran su nombre, ejemplos de valores literales de cada tipo, los operadores que se aplican a ellos y el valor por defecto:

Tipo	Significado	Literales	Operadores	Por defecto
BOOLEAN	Valores lógicos	**true**, **false**	lógicos	false
CHARACTER	Caracteres	'a', '%N'	relacionales	'%U'
INTEGER	Números enteros	-123	+, -, *, ^, //, \\, relacionales	0
DOUBLE	Números reales	123.45e-6	+, -, *, /, ^, relacionales	0.0

En Eiffel los caracteres individuales se escriben entre comillas simples, y las cadenas de texto (clase STRING) entre comillas dobles. El símbolo de porcentaje (%) va siempre seguido de una letra y sirve para indicar un carácter especial:

%N	Nueva línea
%U	Carácter nulo
%%	Carácter porcentaje

3. 2. Declaración de variables

La declaración de variables locales se realiza justo después de la cabecera de un método, con un esquema similar al de Pascal (salvo que la palabra reservada es **local** en lugar de **var**).

También es posible declarar a variables asociadas a la clase (atributos). Estas variables son accesibles desde cualquier método de la clase, y por lo tanto se asemejan a las variables globales de Pascal. En la asignatura no vamos a hacer uso de este tipo de declaraciones "globales".

En Eiffel todas las variables se inicializan automáticamente a un valor por defecto (ver tabla de tipos simples).

3. 3. Sentencia de asignación

asignación ::= *variable* := *expresión*

La sentencia de asignación es idéntica en Eiffel y Pascal. Se debe tener en cuenta que el resultado de la expresión debe ser del mismo tipo (o compatible) que el de la variable.

3. 4. Expresiones y operadores

Las expresiones en Eiffel se forman de la misma manera que en Pascal. Se debe tener en cuenta que (al igual que Pascal) la comprobación de tipos es estricta, y salvo la conversión de INTEGER en DOUBLE, que se realiza automáticamente en el contexto adecuado, el resto de tipos son incompatibles entre sí (no es posible, por ejemplo, comparar un valor de tipo CHARACTER con un valor de tipo INTEGER). A diferencia de Pascal, en Eiffel no existe una construcción que permita

transformar un tipo en otro (moldeado de tipos, *typecasting*). Es necesario llamar al método adecuado para que realice la conversión. Los operadores más importantes son:

Tipo	Operadores	Valores	Resultado
Aritméticos	+, -, *	INTEGER, DOUBLE	Mismo tipo
	// (Cociente), \\ (Resto)	INTEGER	INTEGER
	/ (División)	INTEGER, DOUBLE	DOUBLE
	^ (Potencia)	base INTEGER, DOUBLE exponente INTEGER	Mismo tipo que base
Relacionales	=, /=, <, >, <=, >=	CHARACTER, INTEGER, DOUBLE	BOOLEAN
Lógicos	**not**, **and**, **or**, **and then,** **or else**, **implies**	BOOLEAN	BOOLEAN
Cadenas	+ (Concatenación)	STRING	STRING

- Los operadores // y \\ son equivalentes a **div** y **mod** en Pascal. El operador / representa la división real, el resultado es *siempre* de tipo DOUBLE, incluso cuando sus operandos son enteros y la división da un resultado exacto.
- El operador ^ debe tener un exponente entero. Las potencias con exponentes reales se calculan utilizando el método **pow** de la clase DOUBLE.
- Los operadores lógicos **and** y **or** son estrictos, lo que significa que siempre *evalúan* sus operandos antes de proporcionar un resultado. Los operadores **and then**, **or else** e **implies** son semi-estrictos, y *no evalúan* el segundo operando cuando no es necesario conocer su valor para proporcionar un resultado:
  - *a* **and then** *b* no evalúa el operando *b* si el valor de *a* es **false**. (El resultado es **false** en ese caso). Si el valor de *a* es **true** se comporta igual que el operador **and**.
  - *a* **or else** *b* no evalúa el operando *b* si el valor de *a* es **true**. (El resultado es **true** en ese caso). Si el valor de *a* es **false** se comporta igual que el operador **or**.
  - *a* **implies** *b* no evalúa el operando *b* si el valor de *a* es **false**. (El resultado es **true** en ese caso). Si el valor de *a* es **true** el resultado es el valor del operando *b*.
- El operador de concatenación de cadenas es idéntico al de Pascal.

4. ESTRUCTURAS DE CONTROL

Eiffel dispone de los tres tipos de estructuras de control de la programación estructurada: Secuencial, Alternativa y Repetitiva. Todas las estructuras comienzan por una palabra reservada y se cierran con la palabra reservada **end**, y por lo tanto no es necesario agrupar sentencias mediante construcciones como **begin ... end** en Pascal o **{ }** en C. De esta forma se evitan ambigüedades y la estructura secuencial consiste únicamente en escribir sentencias separadas (opcionalmente) por punto y coma.

4. 1. Estructura secuencial

sentencias ::= *sentencia* { [**;**] *sentencia* }

4. 2. Estructura alternativa

alternativa ::= **if** *condición* **then** *sentencias* { **elseif** *condición* **then** *sentencias* } [**else** *sentencias*] **end**

La estructura alternativa es muy parecida a su equivalente en Pascal, salvo que permite escribir alternativas anidadas (alternativas donde se deben realizar varias acciones distintas si se cumplen condiciones excluyentes) de forma más compacta usando cero o más partes **elseif** en la estructura. Ejemplos:

```
if n < 0 then
  if m < 0 then
    m := -m
  end
else
  n := -n
end
```

```
if den /= 0 then
  cociente := num//den
  resto := num\\den
else
  print("Error!%N");
end
```

```
if ch = 'A' then
  print("avanzar");
elseif ch = 'R' then
  print("retroceder");
elseif ch = 'M' then
  print("mantener");
else
  print("seguir");
end
```

4. 3. Estructura alternativa múltiple

alternativa_múltiple ::= **inspect** *expresión* { **when** *rango* **then** *sentencias* } [**else** *sentencias*] **end**

rango ::= *literal* | *literal* **..** *literal* | *rango* { , *rango* }

Esta estructura es equivalente a la estructura **case** de Pascal. La única diferencia reside en que los rangos deben ser disjuntos y que se produce un error si la expresión produce un valor que no pertenece a ningún rango y no existe parte **else**.

```
c := io.last_character;
inspect c.to_upper
  when 'A'..'Z','Ñ' then
    io.put_string("letra")
  when '0'..'9' then
    io.put_string("digito")
  else
    io.put_string("otra cosa")
end
```

4. 4. Estructura repetitiva

iteración ::= **from** *inicial* [**invariant** asertos] [**variant** *expresión_entera*] **until** *condición* **loop** *sentencias* **end**

A diferencia de Pascal, el lenguaje Eiffel sólo dispone de una única estructura repetitiva, que se corresponde con un bucle con salida al principio donde la condición expresa cuando se debe **salir** del bucle (en Pascal sería equivalente a un bucle **while** con la condición negada). La parte denominada *inicial* en la definición representa una serie de sentencias que se ejecutan justo antes de entrar al bucle (puede estar vacía).

Las partes opcionales invariante y variante del bucle se utilizan en verificación formal de programas. En cada iteración del bucle las condiciones del invariante deben cumplirse, y la expresión entera del variante debe ser menor que el valor obtenido en la iteración anterior.

```
-- calcula el cociente de dos números naturales

division_natural(num,den: INTEGER) : INTEGER is
require
  parametros_positivos : num > 0 and den > 0
local
  cociente, resto : INTEGER
do
  from
    cociente := 0;
    resto := num;
  invariant
    num = cociente*den + resto
  variant
    resto
  until resto < den loop
    resto := resto - den;
    cociente := cociente + 1
  end -- loop
  Result := cociente
ensure
  definicion:  num = cociente*den + resto;
  rango_resto: 0 <= resto and resto < den;
end -- division_natural
```

```
-- escribe los números
-- del 1 al 10

from i := 1
until i > 10 loop
  io.put_integer(i);
  i := i+1
end
```

```
-- detecta si n es primo
-- n es mayor que 2

from
  d := 2
until d >= n or n\\d = 0
loop
  d := d+1
end
es_primo := d >= n
```

4. 5. Aserciones

aserción ::= **check** *asertos* **end**

Las aserciones se utilizan para comprobar en cualquier punto del programa si se cumplen o no una serie de condiciones. Si las condiciones no se cumplen, se genera un error. Por lo tanto esta estructura de control sólo se usa para aquellos casos en que si no se cumplen las condiciones el método no puede realizar su tarea. Más adelante se verá con más detalle su utilización en el apartado de gestión de errores.

```
test is
local ch : CHARACTER
do
  io.put_string("Escriba el caracter [A] o [B]%N");
  io.put_string("Si escribe otro se producira un error: ");
  io.read_character;
  ch := io.last_character;
  check obediente: ch = 'A' and ch = 'B' end
  io.put_string("Muy bien.%N")
end -- test
```

5. PROBLEMAS

- Realizar un programa que adivine en qué número (entre 1 y 1000) está pensando el usuario mediante preguntas cuya respuesta sea si o no.
- Desarrollar los métodos necesarios para calcular la función de Euler: Esta función recibe un entero positivo, *n*, y devuelve el número de enteros menores que cumplen ser primos respecto a *n*. Un número es primo respecto a otro si su máximo común divisor es 1.