**Diseño de Lenguajes de Programación**
3º Curso de Ingeniería Informática del Software
Escuela de Ingeniería Informática (Oviedo)
Universidad de Oviedo

# Introducción

## Objetivo

La práctica obligatoria de la asignatura consiste en el diseño e implementación de un lenguaje de programación aplicando las técnicas de construcción de Procesadores de Lenguajes vistas a lo largo de las clases teóricas de la asignatura.

## Ámbito de este enunciado

Cada grupo de prácticas tiene variaciones en el lenguaje que debe entregar. Si el alumno no ha adquirido este documento en su grupo de prácticas no debe usar este documento como especificación del lenguaje de su práctica. El alumno es responsable de realizar la práctica del grupo al que pertenece, asegurándose de tener la especificación y el programa de ejemplo que le corresponde. No se aceptarán prácticas de un grupo que no le corresponde al alumno.

# Requisitos de implementación

La práctica deberá realizarse de manera individual no permitiéndose prácticas realizadas en grupo.

La práctica se realizará en el lenguaje Java. Queda a decisión del alumno la elección de la plataforma (***Windows***, ***Unix***, ***Mac***, ...) y el entorno de desarrollo para la construcción de la práctica (***Eclipse***, ***Netbeans***, ***Visual Studio Code***, etc.). Deberá implementarse utilizando las técnicas, metalenguajes y herramientas de construcción de compiladores vistas en clase (*ANTLR4, VGEN* y *MAPL*).

El alumno deberá añadir ***únicamente*** las características del lenguaje solicitadas. Cualquier añadido al lenguaje no solicitado (sentencias, operadores, etc.) puntuará negativamente o incluso podría invalidar la práctica.

# Descripción del Lenguaje de entrada al Compilador

## Características del lenguaje

Un programa, en este lenguaje definido para la práctica, estará compuesto por un solo fichero, con una clase "`class`", más un único bloque "`run`" que deberá contener una llamada al constructor que pondrá en marcha la aplicación. Este constructor debe ser uno de los que han sido declarados dentro de la cláusula "`create`" de la clase.

```
class HOLA_MUNDO;
   create
      main; /*Se podrían declarar varios constructores

   feature main is
      do
         print 'H', 'o', 'l', 'a', ' ', 'M', 'u', 'n', 'd', 'o';
         println '!';
   end /*main

end /*HOLA_MUNDO

run main();
```

Una clase está formada por **atributos** (datos en `global`) y **métodos** (código, equivalente a subprogramas en `features`).

A modo de ejemplo se propone el anterior programa para el "Hola Mundo" (Observad que no hay cadenas, por tanto, los caracteres deben ser imprimidos de uno a uno.

Este programa contiene la clase HOLA_MUNDO. El constructor para la clase, de nombre *ejecuta* (declarado en el bloque `create`), invoca `print` y `println`, que son las rutinas de la biblioteca del sistema para escribir un mensaje ("*Hola, mundo!*") al dispositivo de salida.

## Cláusula "*create*"

La cláusula "`create`" declara los nombres de las "`feature`" que pueden ser constructores y que por tanto pueden iniciar la aplicación.

```
class PRUEBA_CREATE;
   créate
      //*Se podrían declarar varios constructores.
      ** La cláusula 'run' podría llamar a cualquiera de ellos.
      *//
      main;
      inicio;
      otroInicio;

   feature main is
         …
   end /*main

   feature inicio is
         …
   end /*main

   feature otroInicio is
         …
   end /*main

end /*PRUEBA_CREATE

run main(); /* Podría ser comentado para iniciar la aplicación con alguno de los siguientes
/* run inicio(); /* Podría ser descomentado para iniciar la aplcación
/* run otroInicio(); /* Podría ser descomentado para iniciar la aplcación
```

El constructor puede tener tipo de retorno, pero no se recogerá el valor devuelto.

Es necesario que haya al menos un constructor. Si hay más de uno, todos los declarados en la cláusula "`create`" deben tener su "`feature`" correspondiente. No puede haber declaraciones en la cláusula "`create`" que no se correspondan con una "`feature`". Por otro lado, en la cláusula "`run`" sólo se puede hacer una invocación a un método que haya sido declarado en la cláusula "`create`".

## Comentarios

Los comentarios de una línea comienzan por "/*" y terminan al final de la línea

```
/* Ejemplo de comentario de una línea
```

Se introduce un comentario de varias líneas que comenzará con "//*" y terminará con "*//".

```
//*
Ejemplo de comentario
de varias líneas
*//
```

## Nombres de variables

Los nombres de los identificadores (variables, métodos, clases, objetos) deben comenzar por una letra y continuar por letras, números o el carácter '_'.

## Aspectos léxicos

- Deberá permitir constantes literales de tipo entero y real. Los literales de tipo real, además de una parte entera, deben tener obligatoriamente algún digito decimal.
- Deberá permitir constantes de tipo carácter (se usarán comillas simples). Se debe admitir la secuencia de escape *'\n'* para representar al carácter de salto de línea.

## Tipos simples o primitivos

- Habrá tres tipos primitivos: entero (INTEGER), real (DOUBLE) y carácter (CHARACTER).

| **Tipo** | **Significado** | **Literales** |
|---|---|---|
| CHARACTER | Caracteres | 'a', '\n' |
| INTEGER | Números Enteros | -123 |
| DOUBLE | Números Reales | 123.45 |

## Definición de variables

- En la misma definición de la variable no se le podrá asignar un valor.

```
i : INTEGER; /*esto estaría bien
j : INTEGER := 2; /*ERROR: esto estaría mal, no se puede inicializar
```

- Se permite la declaración múltiple de variables separadas por comas.

```
i, j, k : INTEGER; /*esto estaría bien
```

- Se permitirá la definición de variables comunes a todos los métodos "`global`" y locales a cada método "`local`" tanto simples como compuestas (arrays y estructuras). La definición de nuevos tipos de estructuras sólo es posible en la zona "`global`" (`deftuple`)). Las declaraciones de tipos deberán ir en la primera parte de "global", dentro de bloque "`types`" y las variables globales irán en un bloque llamado "`vars`"

```
global
   types
      deftuple tuple1 as /*estructura
         a: INTEGER;
         c: CHARACTER;
         d: DOUBLE;
      end
   vars
      t : tuple1; /*estructura
      ai : [2] INTEGER; /*array de una dimensión
      bi : [2][2] INTEGER; /*array de dos dimensiones
```

- Las variables comunes a todos los métodos deberán estar definidas en la etiqueta "`global`" y "`vars`" de la clase y las propias de cada método en la etiqueta "`local`" de cada método.
- No es obligatorio que haya variables globales definidas, en ese caso la etiqueta "`vars`" no es obligatoria, si no hay estructuras nuevas ("`deftuple`") la etiqueta "`types`" no es obligatoria y si no hay variables ni estructuras nuevas, la etiqueta "`global`" no es obligatoria. Lo mismo ocurre con las variables locales, no es obligatorio que las haya y en ese caso tampoco es obligatoria la etiqueta "`local`".
- Todas esas etiquetas podrían existir y estar vacías, esto es podrían existir cualquiera de las etiquetas y que no se haya definido ningún elemento en ella. En el caso de "`vars`" y "`types`" podrían existir y estar vacías, pero en ese caso debe existir la etiqueta "`global`" previa.

```
class
    PRUEBA;
   global
      types
         deftuple tuple1 as /*estructura
             a: INTEGER;
             c: CHARACTER;
             d: DOUBLE;
         end
      vars
         t : tuple1; /*estructura
         dd : DOUBLE; /*valor real
   create
       main;
   feature main is
      local
         ai : [2] INTEGER; /*array de una dimensión
         bi : [2][2] INTEGER; /*array de dos dimensiones
         g, h : INTEGER; /* Son posible declaraciones múltiples de variables
      do
            /* sentencias del método constructor inicio
   end /*main
end /*PRUEBA

run main(); /* Arranque
```

- En de las declaraciones de campos dentro de una "`deftuple`", no se permiten declaraciones múltiples, cada campo debe declararse por separado, aunque tengan tipos comunes.
- En el bloque "`local`" no existen los subbloques "`types`" y "`vars`" porque no es posible crear nuevos tipos locales, sólo globales.

## Asignación

- Sólo se pueden asignar valores de tipos simples.
- No hay promoción automática de tipos.
- No habrá asignación múltiple (por ejemplo '`a := b := 0;`')

```
ai[0] := 7; /*asignación a array de tipo INTEGER
dd := 7.5; /*asignación a variable DOUBLE
ai[3] := 7.5; /*ERROR: asignación de DOUBLE a variable INTEGER
t.a := 5;
t.c := '\n';
t.d := 8.99; /*asignaciones a los campos de una tupla
```

## Conversiones de Expresiones

- No habrá conversiones implícitas.
- Se podrán realizar conversiones explícitas entre los tipos primitivos (*CHARACTER*, *INTEGER* y *DOUBLE*) mediante un operador de conversión (también llamado *cast*). En este caso, el tipo a convertir deberá ser *distinto* al tipo de la expresión. El *cast* se realiza con el operador "*to<TIPO>(EXPRESION)*". Los paréntesis alrededor de la expresión son obligatorios. Ejemplo:

```
t.a := to<INTEGER>(5.6); /*paréntesis obligatorios
```

Aplicaciones del casting:

| | INTEGER | DOUBLE | CHARACTER |
|---|---|---|---|
| **INTEGER** | | SI | SI |
| **DOUBLE** | SI | | |
| **CHARACTER** | SI | | |

El resto de las conversiones directas no están permitidas y deben dar el error correspondiente. Para dichas conversiones hay que hacer conversiones dobles, pasando por *INTEGER*:

```
a : INTEGER; /*declaración de variable INTEGER
f : DOUBLE; /*declaración de variable DOUBLE
c : CHARACTER; /*declaración de variable CHARACTER
...
a := to<INTEGER>(c); /*Correcto
c := to<CHARACTER>(a); /*Correcto. Puede hacer pérdida de precisión
f := to<DOUBLE>(c); /*ERROR, incorrecto. No se permite esta conversión
f := to<DOUBLE>(to<INTEGER>(c)); /*Correcto. Así debe hacerse
```

## Operadores

Los operadores que deberá incluir el lenguaje son:

- **Asignación**: el operador de asignación será "`:=`".
- **Acceso a campo**: el operador de acceso a campo será el operador punto "`.`".
- **Aritméticos**: "`+`", "`-`", "`*`" y "`/`". Aplicables a *INTEGER* y *DOUBLE*. "*mod*" aplicable sólo a *INTEGER*.
- **Comparación**: mayor (`>`), mayor o igual (`>=`), menor (`<`), menor o igual (`<=`), igual (`=`) y distinto (`<>`). Aplicables a *INTEGER* y *DOUBLE*. Los operadores igual (`=`) y distinto (`<>`) también pueden ser aplicados al tipo *CHARACTER*.
- **Lógicos**: "*and*", "*or*" y "*not*". Aplicables sólo a enteros y con evaluación sin corte.

```
i : INTEGER; /*declaraciones
j : INTEGER;
c : CHARACTER;
…
/*Uso de diferentes operadores
if (i < j) and not (c = 'n') then
…
```

## Sentencias de control de flujo

- Sentencia condicional (`else` es opcional y las condiciones no es obligatorio que vayan entre paréntesis). Se puede anidar tantas veces como sea necesario.

```
if <condición> //* condición *// then
   … //* instrucciones *//
else
   … //* instrucciones *//
end
```

- Sentencia "`from…until…loop`". Esta sentencia permite hacer bucles. Al revés que con las sentencias del tipo "`while`", la sentencia "`from…until…loop`" ejecuta el interior del bucle mientras la condición sea falsa y rompe el bucle cuando la condición se cumple.

```
from //* inicialización *//
   … //* asignaciones *//
until <condición> loop
   /* condición que debe ser verdadera para terminar el bucle
   //* instrucciones *//
end
```

- La inicialización puede estar vacía y en ese caso se puede eliminar la palabra `from`. Si existe la cláusula `from`, ésta debe tener a menos una inicialización o asignación de un valor a una variable. También puede haber varias variables inicializadas dentro de dicha cláusula.

```
from //* inicialización *//
   a := 5;
   b[7] := factorial(4);
   st.f := 27.5;
   proc(); //* Error, sólo se permiten asignaciones a variables *//
until <condición> loop
   /* condición que debe ser verdadera para terminar el bucle
   //* instrucciones *//
end
```

- Si la cláusula `from` está vacía, el bucle tiene el aspecto del ejemplo que sigue.

```
until <condición> loop
   /* condición que debe ser verdadera para terminar el bucle
   //* instrucciones *//
end
```

## Entrada y Salida

- Sentencias de lectura y escritura por la E/S estándar para todos los tipos primitivos con las instrucciones "**read**" y "**print**" (tipo *CHARACTER* incluido).
- La sentencia de escritura tendrá dos variantes:
  - Si se usa "**print**", la siguiente escritura se realizará justo donde acabó la anterior.
  - Si se usa "**println**", después de la impresión de la lista de valores de las expresiones, se imprimirá automáticamente un salto de línea.

| **Ejemplo** | `print 1;`<br>`print 2;` | `println 1;`<br>`print 2;` | `println 1, 3;`<br>`print 2;` | `print 1, 3;`<br>`println 2;`<br>`print 4;` |
|---|---|---|---|---|
| **Salida** | 12 | 1<br>2 | 13<br>2 | 132<br>4 |

  - En estas dos variantes, ***no es obligatorio que haya una expresión***. En ese caso, "**println**" imprimirá sólo el salto de línea y "**print**" no hará nada. La instrucción "**println**;" es equivalente a "**print** *'\n'*;".

- Se debe permitir tanto la entrada como la salida de múltiples elementos separados por comas

```
read a, b, t.c; /* a, b y t.c pueden ser de diferentes tipos (t es tupla)
print a, t.c, c+3; //* a, t.c y c pueden ser de diferentes tipos y se permiten expresiones *//
println a, b; //* a y b pueden ser de diferentes tipos y la nueva línea se pone detrás de la última
                 expresión*//
```

- Como ya se vio en el primer ejemplo, no existen las cadenas y para imprimir mensajes deben ser enviados carácter a carácter.

```
println 'H', 'o', 'l', 'a', ' ', 'M', 'u', 'n', 'd', 'o', '!';
print 'H', 'o', 'l', 'a', ' ', 'M', 'u', 'n', 'd', 'o', '!', '\n';
        /*Ambas sentencias hacen lo mismo
```

- No se permite leer ni escribir tuplas completas ni *arrays* completos.

## *Arrays*

- Se permitirá declarar *arrays* multidimensionales (tanto de tipos simples como de tipos compuestos).
- En la definición de un *array*, el tamaño es obligatorio y deberá ser una **constante entera**.
- Se permitirá acceder a los elementos de un *array* mediante una sintaxis de indexación. Para acceder al elemento de un *array*, se usarán **expresiones enteras** (es decir, no tienen por qué ser solamente constantes sino cualquier expresión de dicho tipo).

```
deftuple tuple1 as /*estructura
      a: INTEGER;
      c: CHARACTER;
      d: DOUBLE;
   end
...
   ai : [200] INTEGER; /*array de una dimensión
   af : [200][100] DOUBLE; /*array de dos dimensiones
...
   t : tuple1; /*estructura
...
ai[j - 6] := ai[factorial(to<INTEGER>(t.d)) - t.a]];
//*
El resultado en ambos casos de las expresiones entre corchetes debe ser entero
*//
af[3][45] := 3.1415;
```

- No hay que generar código que compruebe el acceso fuera de rango del *array*.

## Estructuras/Registros

- Un tipo estructura solo podrá ser usado debajo de su declaración, esto es, cualquier variable del tipo estructura debe referirse a alguna de las prefinidas previamente en la sección "`global`" y "`types`".
- La sintaxis de las estructuras es la definida en el ejemplo usando la palabra reservada "`deftuple`". Deben declararse las estructuras obligatoriamente en la sección "`global`" y "`types`".
- Se permitirá acceder a los elementos de una estructura mediante el operador punto ".", por ejemplo "*estructura.campo*".
- En una estructura se podrán definir campos tanto de tipo simple como de otros tipos compuestos, siempre que hayan sido definidos previamente en la sección "`global`" y "`types`".
- No es obligatorio que haya algún campo, esto es, una estructura podría estar vacía y se podrán declarar variables de dicha estructura, pero no podrán ser usadas porque no existe ninguna instrucción que las maneje en bloque (sólo se pueden manipular los campos) y no se podrá acceder a ningún campo porque no se han declarado.
- La declaración de un nuevo tipo estructura solo podrá aparecer a nivel global (fuera de funciones y otras estructuras) dentro de la sección "`global`" y "`types`".

```
class PRUEBA;
   global
      types
         deftuple tuple1 as /* estructura
            a: INTEGER;
            c: CHARACTER;
            d: DOUBLE;
            f, g : CHARACTER; /*ERROR, no se permiten declaraciones múltiples
         end
         deftuple tuple2 as /* estructura
            aa: INTEGER;
            tt: tuple1;
         end
      vars
         t2: tuple2;
         t : tuple1; /* estructura
...
t.d := t2.tt.d; /* asignación de campos de la estructura
```

## Métodos (feature)

El lenguaje permitirá la definición e invocación de métodos. En lo que sigue método es lo mismo que "`feature`".

```
feature mitad (dm : DOUBLE //*, más parámetros*//) : DOUBLE //* Tipos de retorno*// is
   local
```

```
      //* declaraciones locales *//
   do
      //* sentencias *//
end /* Fin de la feature
```

Una "feature" comenzará con la palabra reservada "feature" a la que siguen, si corresponde, entre paréntesis la definición de los parámetros y sus tipos. Finalmente, tras ":" el tipo de retorno.

```
feature mitad (dm : DOUBLE) : DOUBLE is /* Tiene un argumento y valor de retorno
      //* … resto de feature *//
feature mayor (d1 : DOUBLE, d2 : DOUBLE) : DOUBLE is /* Tiene dos argumentos y valor de retorno
      //* … resto de feature *//
feature procedimiento (dm : DOUBLE) is /* Tiene un argumento pero no valor de retorno
      //* … resto de feature *//
feature procedimiento2 is /* No tiene argumentos ni valor de retorno
      //* … resto de feature *//
feature aleatorio : DOUBLE is /* No tiene un argumentos pero sí valor de retorno
      //* … resto de feature *//
```

Tras la cabecera de la función se pueden declarar variables locales dentro de la sección "local". Esta sección puede existir o no existir, pero si hay declaraciones de variables, entonces "local" debe existir previamente.

```
feature procedimiento1 (dm : DOUBLE) is /* Tiene un argumento pero no valor de retorno
   local /* Puede existir y estar vacía
      //* No hay declaraciones, sigue el resto de feature *//
feature procedimiento2 (dm : DOUBLE) is /* Tiene un argumento pero no valor de retorno
      /* local /* No hay declaraciones
      //* … resto de feature *//
feature procedimiento3 (dm : DOUBLE) is /* Tiene un argumento pero no valor de retorno
   local
      i : INTEGER; /* Declaraciones
      //* … resto de feature *//
feature procedimiento3 (dm : DOUBLE) is /* Tiene un argumento pero no valor de retorno
   /* local
      i : INTEGER; /* Error, si existe una variable debió colocarse previamente "local"
      //* … resto de feature *//
```

Tras la declaración de las variables locales, la cláusula "do" es seguida de las sentencias que componen la "feature".

- Sólo dentro de los métodos puede haber sentencias (no puede haber sentencias a nivel global, a excepción de la llamada al método de inicio desde el bloque "run").
- Todos los parámetros se pasan por valor.
- Los parámetros y el valor de retorno solo podrán ser de tipos simples.
- No habrá conversiones implícitas de los tipos de los argumentos a los tipos de los parámetros.
- Tampoco habrá conversión implícita del tipo del valor de retorno al tipo de retorno del método.
- Si un método tiene tipo de retorno, se deberá comprobar que sus "**return**" tengan una expresión del mismo tipo. Si no tiene tipo de retorno, sus "**return**" no deben tener valor de retorno.

```
return i; /* Estará en un método (feature) con valor de retorno
return; /* Estará en un método (feature) sin valor de retorno (retorna void)
```

- Un método podrá declarar variables locales sólo al inicio de su cuerpo en la sección "local". Por tanto, no podrá haber declaración de variables locales en cualquier lugar del método ni dentro de bloques anidados (por ejemplo, no podrá haber dentro de un "from…until…loop").
- Un método podrá invocarse a sí mismo mediante recursividad directa.
- Es obligatorio que haya métodos en la clase de un fichero fuente. También es obligatorio que exista el "run". Si un método tiene valor de retorno, debe tener obligatoriamente la instrucción "**return**".
- No habrá sobrecarga de métodos.
- Un método solo podrá invocarse después de haber sido definido. Por tanto, no podrá llamarse a un método definido más abajo en el fichero fuente.

- No es obligatorio que un método tenga sentencias, a excepción del "**return**" en caso de tener valor de retorno.
- Un método, aunque que tenga valor de retorno, podrá ser invocado como si fuera una sentencia, esto es, sin recoger el valor retornado. Dicho valor simplemente se ignorará.
- El bloque "run" debe ser el último del fichero, después de la definición de la clase.

```
class PRUEBA;
   global
      vars
         dd : DOUBLE; /*valor real
   create
       main;

   feature mitad (dm : DOUBLE) : DOUBLE is
      //* declaración con parámetros DOUBLE y valor de retorno DOUBLE *//
      /* local /* no hay declaraciones
      do
         return dm / 2;
   end /* mitad

   feature main (i : INTEGER) is
      /* local /* no hay declaraciones
      do
         dd := to<DOUBLE>(i); /* casting a double
         dd := mitad (dd); /* se recoge el valor de retorno
         mitad (dd); /* No se recoge el valor de retorno
   end /* main
end /* PRUEBA

run main(2);
```

### Programas de Ejemplo

Entre el material entregado al alumno hay dos programas de ejemplo en este lenguaje:

- Revisar los programas.
- En principio muchos de los programas entregado ahora no tienen una funcionalidad clara, pero deben ser reconocidos por el léxico y el sintáctico.

## Entregables

### Parciales

Para los alumnos en evaluación continua habrá una entrega parcial que se anunciará en clase. Los objetivos de esta entrega también serán comunicados en clase.

### Entrega final

La práctica finalizada se entregará directamente el día del examen, no siendo necesaria una entrega previa.

Al examen, además de los códigos fuente de la práctica, habrá que llevar **obligatoriamente** los documentos con las especificaciones de las distintas fases del compilador en formato PDF. En concreto, las especificaciones que se piden son:

- Léxico del lenguaje expresado mediante Expresiones Regulares.
- Sintaxis del lenguaje mediante una Gramática Libre de Contexto.
- Descripción de los nodos del Árbol Abstracto (AST) mediante una Gramática Abstracta.
- Descripción de la fase de comprobación de tipos del análisis semántico mediante una Gramática Atribuida.
- Descripción de la fase de Selección de Instrucciones mediante una Especificación de Código (plantillas de código).

El hecho de no entregar dichos documentos o hacerlo de forma errónea o incompleta supondrá no superar el examen práctico.