# Procesamiento de Lenguajes (PL)
# Curso 2019/2020

## Práctica 3: traductor descendente recursivo

### Fecha y método de entrega

La práctica debe realizarse de forma individual, y debe entregarse a través del servidor de prácticas del DLSI **antes de las 23:59 del 26 de abril de 2020**. Los ficheros fuente en Java se comprimirán en un fichero llamado "`plp3.tgz`", y no debe haber directorios (ni `package`) en él, solamente los fuentes en Java.

Al servidor de prácticas del DLSI se puede acceder de dos maneras:

- Desde la web del DLSI (http://www.dlsi.ua.es), en el apartado "Entrega de prácticas"
- Desde la URL http://pracdlsi.dlsi.ua.es

Una vez en el servidor, se debe elegir la asignatura PL y seguir las instrucciones.

### Traducción

La práctica consiste en implementar un traductor descendente recursivo basado en la práctica 1, que traduzca del lenguaje fuente de la práctica 1 a un lenguaje parecido al C. Debes considerar los siguientes aspectos al realizar la práctica:

1. **MUY IMPORTANTE:** diseña el ETDS en papel, con la ayuda de algún árbol o subárbol sintáctico, pero no es necesario que uses árboles completos, el proceso de traducción tiene varias partes diferentes (prefijos, expresiones, ámbitos) que se pueden resolver por separado, y luego juntar. Cuando tengas diseñado el ETDS (o alguna parte), puedes empezar a transformar el analizar sintáctico en un traductor, que irá construyendo la traducción mientras va analizando el programa de entrada.

2. Para utilizar una variable o parámetro en una asignación o en una expresión debe haberse declarado con anterioridad; en caso contrario se emitirá un error semántico, como se describe a continuación.

3. Las variables y funciones deben llevar un prefijo con el nombre de su función o clase padre (que, a su vez, puede llevar un prefijo con el nombre de su padre, etc). Además, las variables declaradas en bloques interiores llevan un prefijo con "`_`" que depende del nivel de anidamiento del bloque en el que se han declarado (como se puede ver en el ejemplo). **IMPORTANTE:** debes usar atributos heredados para resolver este problema. Los parámetros de las funciones no llevan prefijo.

4. En las expresiones aritméticas y relacionales se pueden mezclar variables/parámetros, números enteros y reales; cuando aparezca una operación con dos operandos enteros, se generará el operador con el sufijo "`i`"; cuando uno de los dos operandos (o los dos) sea real, el sufijo será "`r`". Si se opera un entero con un real, el entero se convertirá a real usando "`itor`", como en el ejemplo. Como en C/C++, la división será de tipo entero si ambos operandos son enteros, y real en otro caso.

5. En el caso de los operadores aritméticos (suma/resta y producto/división), en la sesión 2 del tema de traducción dirigida por la sintaxis se explica como obtener el tipo y la traducción.

6. El resultado de un operador relacional es siempre de tipo entero, independientemente del tipo de los operandos), aunque el operador que se debe generar llevará el sufijo "`i`" o el sufijo "`r`" según el tipo de los operandos. Por ejemplo, `2.5 * (2 == 2.5)` se traduciría por `2.5 *r itor( ( itor(2) ==r 2.5 ) )` . En este caso, la traducción de la expresión entre paréntesis (el operador "`==`"), al ser de tipo entero y tener que multiplicarse por `2.5`, debe convertirse a real con `itor`.

7. En las asignaciones pueden darse tres casos:

   - La variable es real y la expresión es entera: en ese caso, la expresión debe convertirse a real con "`itor`"
   - La variable y la expresión son del mismo tipo: en ese caso no se genera ninguna conversión

- La variable es de tipo entero y la expresión es real: se debe producir un error semántico de tipos incompatibles, como se describe a continuación.

Estas comprobaciones deben realizarse al final de la regla, después de haber obtenido la traducción y el tipo de la expresión.[1]

8. La expresión que va a continuación de "`if`" debe ser de tipo entero; esta comprobación debe hacerse después de la expresión, antes o después de procesar el "`:`", da igual, pero antes del final de la regla.

9. En los bloques es posible declarar nuevas variables con el mismo nombre que otros símbolos (parámetros, variables, funciones, clases) en ámbitos exteriores, por lo que tendrás que utilizar una tabla de símbolos con gestión de ámbitos. A estos efectos, los ámbitos se corresponden con los bloques entre llaves, es decir, comienzan con "`{`" y terminan con "`}`", excepto en el caso de los parámetros de las funciones, que pertenecen al ámbito que comienza justo después (el que contiene el código de la función), por lo que no es posible declarar variables locales dentro de la función con el mismo nombre que los parámetros (excepto si se hace dentro de otro bloque entre llaves).

**Ejemplo**

```
class Main {
  fun main  int a;float b {

    fun submain float c {
      print 1+2-(3*4.5/6);
      float d;
      d = a+b/c;
      if a<d :
        float e
      else {
        int b;
        int d;
        b = a+77;
        d = b+33;
        print d+c
      }
      fi
    }

    class A {
    }

    class B {
      fun mainB float g {
        {
          int h;

          h = 7;
          {
            float i;

            i = 2.5 != 2.4999
          }
        }
      }
    }

    /* codigo main */

    print 7
  }
}
```

```
// class Main
void Main_main(int a,float b) {
void Main_main_submain(float c) {
  printf("%f",itor(1 +i 2) -r (itor(3) *r 4.5 /r itor(6)));

  float Main_main_submain_d;

  Main_main_submain_d = itor(a) +r b /r c;

  if (itor(a) <r Main_main_submain_d)
    float Main_main_submain_e;
  else
  {
    int Main_main_submain__b;
    int Main_main_submain__d;

    Main_main_submain__b = a +i 77;
    Main_main_submain__d = Main_main_submain__b +i 33;
    printf("%f",itor(Main_main_submain__d) +r c);
  }
} // Main_main_submain

// class Main_main_A


// class Main_main_B
void Main_main_B_mainB(float g) {
 {
  int Main_main_B_mainB__h;

  Main_main_B_mainB__h = 7;
  {
    float Main_main_B_mainB___i;

    Main_main_B_mainB___i = itor(2.5 !=r 2.4999);
  }
 }
} // Main_main_B_mainB

  printf("%d",7);
} // Main_main
```

**Nota:** la traducción de ejemplo está ligeramente modificada para que se vea mejor, no es importante que tenga exactamente los mismos espacios en blanco y saltos de línea.

[1] La comprobación del error semántico se podría hacer pasando el tipo de la variable como atributo heredado a las expresiones, pero se complica mucho el código y no vale la pena, es mejor esperar al final de la regla.

## Mensajes de error semántico

Tu traductor ha de detectar los siguientes errores de tipo semántico (en todos los casos, la fila y la columna indicarán el principio del token más relacionado con el error), y emitir el error lo antes posible:

1. No se permiten dos identificadores con el mismo nombre en el mismo ámbito, independientemente de que sus tipos sean distintos. El error a emitir si se da esta circunstancia será:

   ```
   Error semantico (fila,columna): 'lexema', ya existe en este ambito
   ```

2. No se permite acceder en las instrucciones y en las expresiones a una variable no declarada:

   ```
   Error semantico (fila,columna): 'lexema', no ha sido declarado
   ```

3. No se permite asignar un valor de tipo real a una variable de tipo entero:

   ```
   Error semantico (fila,columna): 'lexema', tipos incompatibles entero/real
   ```

   En este caso, el lexema será el del operador de asignación.

4. En las instrucciones y expresiones solamente se pueden utilizar identificadores de variables o parámetros, no nombres de función ni de clase. En caso contrario, se debe emitir este error:

   ```
   Error semantico (fila,columna): 'lexema' debe ser de tipo entero o real
   ```

5. La expresión de la instrucción "`if`" debe ser de tipo entero,[2] en caso contrario se emitirá este error:

   ```
   Error semantico (fila,columna): 'lexema' debe ser de tipo entero
   ```

## Notas técnicas

1. Aunque la traducción se ha de ir generando conforme se realiza el análisis sintáctico de la entrada, dicha traducción se ha de imprimir por la salida estándar únicamente cuando haya finalizado con éxito todo el proceso de análisis; es decir, si existe un error de cualquier tipo en el fichero fuente, la salida estándar será nula (no así la salida de error, que en el caso de errores léxicos y sintácticos debe ser como la de la práctica 1).

2. Para realizar la práctica lo recomendable es empezar por la gestión de símbolos y ámbitos anidados, luego continuar por la traducción de expresiones e instrucciones, luego funciones y clases, y por último resolver el problema de los prefijos.

3. Para detectar si una variable se ha declarado o no, y para poder emitir los oportunos errores semánticos, es necesario que tu traductor gestione una tabla de símbolos para cada nuevo ámbito en la que se almacenen sus identificadores. En la web de la asignatura se publicarán un par de clases para gestionar la tabla de símbolos. Es aconsejable guardar en la tabla de símbolos el nombre traducido del símbolo, además del nombre original y el tipo, como se explica más adelante.

4. Como en prácticas anteriores, se publicará en la web un autocorrector.

5. La práctica debe tener varias clases en Java:

   - La clase `plp3`, que tendrá solamente el siguiente programa principal (y los `import` necesarios):

     ```
     class plp3 {
         public static void main(String[] args) {

             if (args.length == 1)
             {
               try {
                 RandomAccessFile entrada = new RandomAccessFile(args[0],"r");
                 AnalizadorLexico al = new AnalizadorLexico(entrada);
                 TraductorDR tdr = new TraductorDR(al);
     ```

[2] Recuerda que los operadores relacionales devuelven siempre un valor de tipo entero.

```
            String trad = tdr.S(""); // simbolo inicial de la gramatica
            tdr.comprobarFinFichero();
            System.out.println(trad);
          }
          catch (FileNotFoundException e) {
            System.out.println("Error, fichero no encontrado: " + args[0]);
          }
        }
        else System.out.println("Error, uso: java plp3 <nomfichero>");
    }
}
```

**Nota:** el método del símbolo inicial `S` recibe un parámetro (atributo heredado) que sería el prefijo inicial, que debe ser siempre una cadena vacía (`""`).

- La clase `TraductorDR` (copia adaptada de `AnalizadorSintacticoDR`), que tendrá los métodos/funciones asociados a los no terminales del analizador sintáctico, que deben ser adaptados para que devuelvan una traducción (además de quizá para devolver otros atributos y/o recibir atributos heredados), como se ha explicado en clase de teoría. Si un no terminal tiene que devolver más de un atributo sintetizado, será necesario utilizar otra clase (que podría llamarse `Atributos`) con los atributos que tiene que devolver, de manera que el método asociado al no terminal devuelva un objeto de esta otra clase.
- Las clases `Token` y `AnalizadorLexico` del analizador léxico.
- Las clases `Simbolo` y `TablaSimbolos` publicadas en la web, que no es necesario modificar (y por lo tanto se recomienda no hacerlo):
  - La clase `Simbolo` tiene solamente los atributos (públicos, por simplificar) y el constructor. Para cada símbolo se almacena su nombre en el programa fuente, su tipo (entero, real, clase o función) y su traducción al lenguaje objeto, que se debe obtener en la declaración del símbolo, y se usará en las expresiones (en el caso de variables, por supuesto).
  - La clase `TablaSimbolos` permite la gestión de ámbitos anidados (utilizando una especie de pila de tablas de símbolos), y tiene dos métodos para añadir un nuevo símbolo, y para buscar identificadores que aparezcan en el código.

## Ámbitos anidados

Para gestionar los ámbitos anidados, se recomienda:

1. En la clase `TraductorDR`, declarar un objeto de la clase `TablaSimbolos` (que se podría llamar `tsActual`, por ejemplo) que será la tabla de símbolos en la que se guarden todos los símbolos en un momento dado. Este objeto se debe crear (`new`) en el constructor de `TraductorDR`, usando `null` como parámetro del constructor de `TablaSimbolos` (porque inicialmente no hay ámbito anterior).

2. Cuando comience un nuevo ámbito (p.ej. después de "`{`" en la regla de "`class`"), se debe crear una nueva tabla de símbolos, guardando como tabla del ámbito anterior la tabla del ámbito actual (`tsActual`), y haciendo que esa nueva tabla pase a ser la tabla actual:

   ```
   tsActual = new TablaSimbolos(tsActual);
   ```

   De esta manera, usando la referencia a la tabla padre, se construye una lista enlazada de tablas de símbolos que funciona como una pila (sólo se inserta/extrae por el principio de la lista).

3. Cuando el ámbito se cierra (p.ej. después de "`}`" en la regla de "`class`"), se restaura la tabla de símbolos anterior (se *desapila*):

   ```
   tsActual = tsActual.getPadre();  // recupera la tabla del ámbito anterior
   ```

Ejemplo:

```
/* ámbito 1 (ámbito inicial, creado en el constructor) */

class A  /* comienza ámbito 2 (padre=1) */   {
  fun funcion   /* comienza ámbito 3 (padre=2) */  int a;float b {
     print a+b
  }                     /* fin ámbito 3 (restaura ámbito 2) */

  class B               /* comienza ámbito 4 (padre=2) */  {

    fun otra /* comienza ámbito 5 (padre=4) */   float c {
      float d;          /* c y d van en el mismo ámbito, el 5 */
      print d*c;

      if 2<d :
        float e         /* e va al ámbito 5, con c y d */

      else {            /* comienza ámbito 6 (padre=5) */
        int b;
        int d;
        b = 2+77;
        d = b+33;
        print d+c
      }                /* fin ámbito 6 (restaura ámbito 5) */
      fi
    }                  /* fin ámbito 5 (restaura ámbito 4) */
  }                    /* fin ámbito 4 (restaura ámbito 2) */
}                      /* fin ámbito 2 (restaura ámbito 1) */
```

La traducción de este ejemplo sería:

```
// class A
void A_funcion(int a,float b) {
  printf("%f",itor(a) +r b);

} // A_funcion

// class A_B
void A_B_otra(float c) {
 float A_B_otra_d;

 printf("%f",A_B_otra_d *r c);

 if (itor(2) <r A_B_otra_d)
   float A_B_otra_e;
 else
 {
   int A_B_otra__b;
   int A_B_otra__d;

   A_B_otra__b = 2 +i 77;
   A_B_otra__d = A_B_otra__b +i 33;

   printf("%f",itor(A_B_otra__d) +r c);
 }
} // A_B_otra
```