Procesamiento de Lenguajes (PL)
Curso 2016/2017

Práctica 3: traductor descendente recursivo

Fecha y método de entrega

La práctica debe realizarse de forma individual, como las demás prácticas de la asignatura, y debe entregarse a través del servidor de prácticas del DLSI **antes de las 23:59 del miércoles 12 de abril de 2017**.

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 (similar a C++) a un lenguaje parecido en castellano.

Ejemplo

```
class A {
  public:
    int f1(int n,float s);
    float ff1(float r) {
      int i;

      i = 2*3;
      {
        int r;

        r = 35/7;
        i = i - r + 4;
      };
      return i+r;
    }
  private:
    class B {
      public:
        int ff2(int a,int b) {
          return a+b*b;
        }
      private:
        float f2 (float r,float s,float t);
        class C {}
    }
}
```

```
clase A {
 publico A::f1 (n:entero x s:real -> entero);

 publico A::ff1 (r:real -> real)
 {
  var i:entero;

  i := 2 *i 3;
  {
   var r:entero;

   r := 35 /i 7;
   i := i -i r +i 4;
  }
  retorna itor(i) +r r;
 }

 privado clase A::B {
  publico A::B::ff2 (a:entero x b:entero -> entero)
  {
   retorna a +i b *i b;
  }

  privado A::B::f2 (r:real x s:real x t:real -> real);

  privado clase A::B::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. Solamente cuando lo tengas diseñado puedes empezar a transformar el analizar sintáctico en un traductor, que irá construyendo la traducción mientras va analizando el programa de entrada.

2. 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.

3. Las clases y métodos en el lenguaje objeto llevarán un prefijo con el nombre de las clases que los contienen, separando cada nombre con "`::`", como se puede ver en el ejemplo.

4. Los métodos, tengan o no código, deben llevar un prefijo que indique si son privados o públicos. La traducción de los parámetros y sus tipos es la que se indica en el ejemplo; el tipo que devuelve el método aparece después del símbolo "`->`"

5. Cada vez que se abre un bloque de código se debe abrir un nuevo ámbito, excepto en el caso de los métodos, en los que el ámbito se abre con el primer parámetro, y las variables declaradas al nivel del bloque del método deben estar en el mismo ámbito. Sólo se abre un nuevo ámbito en un bloque que aparezca dentro del bloque del método. En el método `ff1` del ejemplo, el parámetro `r` y la variable `i` están en el mismo ámbito, y se abre un nuevo ámbito con el bloque, lo que permite declarar otra variable con el nombre `r` que *oculta* el parámetro `r` (de tipo real).

6. En las expresiones aritméticas se pueden mezclar variables, parámetros y números enteros y reales; cuando aparezca una operación con dos operandos enteros, se generará el operando 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.

7. Tanto en las asignaciones como en la instrucción "`return`" pueden darse tres casos:

   - La variable (o el método, si es un "`return`") es real y la expresión es entera: en ese caso, la expresión debe convertirse a real con "`itor`"
   - La variable o método y la expresión son del mismo tipo: en ese caso no se genera ninguna conversión
   - La variable o método es de tipo entero y la expresión es real: se debe producir un error semántico de tipos incompatibles, como se describe a continuación.

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 de la aparición incorrecta del token):

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. Los identificadores de las instrucciones tienen que ser variables simples en el lenguaje fuente. Si corresponden a clases o métodos, el error a emitir será:

   ```
   Error semantico (fila,columna): 'lexema' no es una variable
   ```

4. No se permite asignar un valor de tipo real a una variable de tipo entero, ni devolver un valor real en un método entero:

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

   En este caso, el lexema será el del operador de asignación o el del "`return`".

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).

2. 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.

3. 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);

            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>");
    }
}
```

   - 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 (o tiene atributos heredados), será necesario utilizar otra clase (que debe 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