Procesamiento de Lenguajes (PL)
Curso 2024/2025

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 miércoles 16 de abril de 2025**. 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 desde la web del DLSI (https://www.dlsi.ua.es), en el apartado "Entrega de prácticas". 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 al Pascal.

Ejemplo

```
int f(int a,float &b)
{
  int i;
  int j;

  if (2.5+3+4+i+a)
  {
    float i = 7;

    {
      int i = 7+2;

    }
    float j;

    j = 1+2+3+4.0+i;
    {
      float a;int b;

      a = 1+2;
    }
    {
      {
        int j;

        j = 1+2-3;
      }
    }
    if (1)
      int k;
    {
      int e;
      float g;
    }
  }
}
```

```
function f(a:integer;var b:real):integer;
var i:integer;
    j:integer;
    f_i:real;
    f__i:integer;
    f_j:real;
    f__a:real;
    f__b:integer;
    f___j:integer;
    f_k:integer;
    f__e:integer;
    f__g:real;
begin
    if 2.5 +r itor(3) +r itor(4) +r itor(i) +r itor(a) then
      begin
       f_i := itor(7);
       begin
         f__i := 7 +i 2
       end;


       f_j := itor(1 +i 2 +i 3) +r 4.0 +r f_i;
       begin

         f__a := itor(1 +i 2)
       end;
       begin
         begin

           f___j := 1 +i 2 -i 3
         end
       end;
       if 1 then
         doNothing;
       begin
         doNothing

       end
     end
end
```

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. Si la función no tiene parámetros, en la traducción no se generarán los paréntesis:

```
float funcion()                 function funcion:real;
...
```

4. En el lenguaje objeto (similar a Pascal), las variables sólo se pueden declarar al principio, antes del código, por lo que las traducciones de las declaraciones de variables del lenguaje fuente deben agruparse en un atributo diferente al utilizado para traducir el código.

5. Además, las variables declaradas dentro de bloques deben traducirse poniendo como prefijo el nombre de la función y uno o más subrayados, según el nivel de anidamiento del bloque.[1] En caso de que exista más de una variable con el mismo nombre (permitido en C/C++), se considerará que es la variable del ámbito abierto más cercano a donde se utiliza.

6. El nombre de la función no debe almacenarse en la tabla de símbolos, pero los argumentos sí deben almacenarse (se consideran como variables locales especiales).

7. En el lenguaje fuente, el "`;`" se usa al final de cada instrucción de asignación o declaración de variable; en el lenguaje objeto, el "`;`" sirve para separar dos instrucciones en una secuencia.

8. Una declaración de variable con inicialización se traduce como una declaración seguida de una asignación. La variable debe guardarse en la tabla de símbolos **antes** de procesar la expresión de la inicialización.

9. Si la instrucción del `if` es solamente una declaración de variable sin inicialización, el código traducido será `doNothing`, la traducción de la declaración se colocará con las demás traducciones de declaraciones, y la variable se guardará en la tabla de símbolos (sin abrir un nuevo ámbito). De forma similar, si la secuencia de instrucciones de un bloque sólo contiene declaraciones, se generará la instrucción `doNothing`, y las declaraciones se traducirán como en cualquier bloque con instrucciones que no sean declaraciones.

10. En las expresiones aritméticas se pueden mezclar variables, 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.

11. En las asignaciones e inicializaciones 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.

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
```

[1] Lo más sencillo es almacenar la traducción de cada variable en la tabla de símbolos

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.

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. En el Moodle 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 de los símbolos, además del nombre original y el tipo, como se explica más adelante.

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.Fun(); // 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
   - Las clases `Simbolo` y `TablaSimbolos` publicadas en el Moodle de la asignatura, 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) y su traducción al lenguaje objeto, que se usará con los identificadores declarados en el programa fuente, y se debe construir en la declaración del identificador para que se use en la traducción de la propia declaración y en la traducción de las expresiones.

- La clase `TablaSimbolos` permite la gestión de ámbitos anidados (utilizando una especie de pila de tablas de símbolos), y tiene métodos para añadir un nuevo símbolo y para buscar identificadores que aparezcan en el código.