Procesamiento de Lenguajes (PL)
Curso 2023/2024

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 2024**. Los ficheros fuente en Java tienen que poderse compilar con la versión de Java instalada en los laboratorios, no deben tener ningún "`package`", y deben comprimirse (sin directorios) en un fichero llamado "`plp3.tgz`" (aunque el servidor renombra cualquier `.tgz` a ese nombre).

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 ALC) a un lenguaje parecido al C.

Ejemplo

```
funcion prueba;

   funcion hija;
   blq
       var c : real; fvar
       hija := 7.555
   fblq
blq
 var a : entero;
     b : tabla [ 1..7, 0..2 ] de real;
     c : puntero de puntero de entero;
     d : real;
     e : tabla [ 1..5 ] de tabla [ 3..4 ] de
           puntero de entero;
 fvar

   d := 7 % a / 2 * 2.3;
   escribe ( d + 2 // 3 - 4.5 );
   blq
     var c:real; fvar
     c := 256 % 2
   fblq;
   blq
     var cc:entero; fvar
     blq
       var d:entero;
           b:real;
       fvar
       b := cc;
       escribe(6);
       prueba := 7+5*4
    fblq
  fblq
fblq
```

```
float hija()
{
  float c;

  return 7.555;
}

float prueba()
{
  int a;
  float b[7][3];
  int** c;
  float d;
  int* e[5][2];

  d = itor(7 % a) /r itor(2) *r 2.3;
  printf("%f",d +r itor(2 /i 3) -r 4.5);
  {
    float _c;

    _c = itor(256 % 2);
  }
  {
    int _cc;

    {
      int __d;
      float __b;

      __b = itor(_cc);
      printf("%d",6);
      return itor(7 +i 5 *i 4);
    }
  }
}
```

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 variables declaradas en bloques `blq-fblq` interiores llevan un prefijo con "`_`" que depende del nivel de anidamiento del bloque en el que se han declarado.[1]

4. En las expresiones aritméticas se pueden mezclar variables, números enteros y reales; cuando aparezca una operación con dos operandos enteros, se generará el operador con el sufijo "`i`" (excepto en el caso del operador "`%`", que no lleva sufijo); 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.

5. En las asignaciones pueden darse cuatro casos:

   - La variable antes de "`:=`" coincide con el nombre de la función (debes usar un atributo heredado para comprobarlo), en cuyo caso se genera una instrucción "`return`" con la expresión a la derecha de "`:=`", que debe convertirse a real con "`itor`" si es entera (ten en cuenta que en el lenguaje objeto las funciones son "`float`")
   - 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.

6. En los bloques `blq-fblq` es posible declarar nuevas variables con el mismo nombre que en ámbitos exteriores (pero nunca con el nombre de la función), por lo que tendrás que utilizar una tabla de símbolos con gestión de ámbitos. El nombre de la función no debe guardarse en la tabla de símbolos, debe pasarse como atributo heredado.

7. En las expresiones y en la asignación sólo es posible utilizar variables de tipos simples (entero o real).[2]

8. En el lenguaje fuente, los operadores `//` y `%` operan solamente con números enteros, y el operador `/` siempre realiza una división real aunque los argumentos sean enteros, es decir, `7/2` es `3.5`.

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 puede declarar una variable con el nombre de la función (que debe pasarse como atributo heredado y no debe guardarse en la tabla de símbolos). El error a emitir si se da esta circunstancia será:

   ```
   Error semantico (fila,columna): en 'lexema', no puede llamarse igual que la funcion
   ```

2. No se permiten dos identificadores con el mismo nombre en el mismo ámbito, independientemente de que sus tipos sean distintos:

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

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

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

[1] Debes usar atributos heredados para resolver este problema.

[2] Sí, las variables de tipo *array* o puntero no se pueden usar y no sirven para nada, pero hay que traducirlas igual (recuerda que esto es una práctica, nada más).

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

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

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

5. En expresiones y asignaciones solamente se pueden utilizar variables de tipo simple:

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

6. Los operadores `//` y `%` necesitan que ambos operadores sean enteros, en caso contrario se debe emitir un error semántico (uno para el operador izquierdo, y otro para el derecho):

   ```
   Error semantico (fila,columna): en 'lexema', el operando izquierdo debe ser entero
   Error semantico (fila,columna): en 'lexema', el operando derecho debe ser entero
   ```

7. En la declaración de tablas, el segundo número de un rango debe ser mayor o igual que el primero:[3]

   ```
   Error semantico (fila,columna): en 'lexema', rango incorrecto
   ```

Por ejemplo, en esta declaración (suponiendo que `variableRepetida` se ha declarado previamente):

```
variableRepetida: tabla [ 3 .. 2 ] de patata;
```

se debería dar el error de que `variableRepetida` ya existe en el ámbito, a continuación (después de corregir ese error) el error de rango incorrecto, y finalmente el error sintáctico en `patata`.

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.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>");
    }
}
```

[3] El lexema, fila y columna deben ser los del segundo número.

- 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, tabla, puntero) 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.

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, 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, se restaura la tabla de símbolos anterior (se *desapila*):

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