Procesamiento de Lenguajes (PL)
Curso 2022/2023

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 viernes 28 de abril de 2023**. 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 (http://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 de un subconjunto de un lenguaje similar a Pascal (con alguna modificación) a un lenguaje parecido a C.

Ejemplo

```
algoritmo colores;
 funcion rojo:entero;
   var a,b,c:entero;
   blq (* rojo *)
     a := 7;
     b := a+3
   fblq;

 funcion verde:real;
   funcion fosfo:entero;
     var j:real;
     funcion rito:real;
       var k:entero;
       blq (* rito *)
         k := 643;
         j := k / 10
       fblq;
     blq (* fosfo *)
       j := 34.5
     fblq;
   var m,n:real;
       o,p:entero;
   blq (* verde *)
     o := (27-20) * 3;
     m := o;
     mientras o>10 hacer
       blq
         si 2*o = 22 entonces
           p := 1
         sino
           p := 0
         fsi;
         o := o - 1
       fblq
   fblq;
 var a,b,c:entero;
 blq (* colores *)
   a := 77;
   escribir(a+3)
 fblq
```

```
// algoritmo colores
int rojo_a,rojo_b,rojo_c;
int rojo() {
  rojo_a =i 7 ;
  rojo_b =i rojo_a +i 3 ;
}

double verde_fosfo_j;
int verde_fosfo_rito_k;
double verde_fosfo_rito()
{
  verde_fosfo_rito_k =i 643 ;
  verde_fosfo_j =r itor(verde_fosfo_rito_k) /r itor(10) ;
}

int verde_fosfo()
{
  verde_fosfo_j =r 34.5 ;
}

double verde_m,verde_n;
int verde_o,verde_p;
double verde() {
  verde_o =i (27 -i 20) *i 3 ;
  verde_m =r itor(verde_o) ;
  while ( verde_o >i 10)
  {
    if ( 2 *i verde_o ==i 22 )
      verde_p =i 1 ;
    else
      verde_p =i 0 ;
    verde_o =i verde_o -i 1 ;
  }
}

int main_a,main_b,main_c;
int main() {
  main_a =i 77 ;
  printf("%d\n",main_a +i 3);
}
```

Como se puede observar en el ejemplo de traducción, hay varios aspectos a considerar:

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. En el lenguaje fuente las funciones pueden tener funciones locales (subprogramas), pero en el lenguaje objeto no se permite. Por tanto, en la traducción los subprogramas deben llevar un prefijo que indique su padre (que a su vez puede llevar un prefijo con el abuelo, bisabuelo, etc.)

3. Las variables declaradas en una función son accesibles para los subprogramas de las funciones, por lo que es necesario en la traducción emitir dichas variables como variables globales, con un prefijo similar al de los subprogramas

4. En las asignaciones y expresiones, el lenguaje fuente permite la sobrecarga de operadores (habitual en muchos lenguajes de programación), es decir, permite usar el mismo operador para sumar dos enteros o para sumar dos reales. Sin embargo, en el lenguaje objeto no se permite la sobrecarga de los operadores (ni la de la asignación), por lo que es necesario usar el operador adecuado (en el caso de la suma se usaría "`+i`" o "`+r`"), y realizar las conversiones necesarias de entero a real usando la función 'itor'.

5. En el lenguaje fuente, como en Pascal, el operador "`/`" siempre denota la división real, por lo que aunque los operandos sean enteros, el resultado será real. Por ejemplo "`1/2`" es `0.5` (en C y otros lenguajes sería 0). Para la división entera se debe usar el operador "`//`" (por ejemplo, "`7 // 2`" es `3`). Este operador no se puede utilizar si cualquiera de los operandos no es de tipo entero.

6. En el lenguaje fuente, cada subprograma constituye un nuevo ámbito, y las reglas de ámbitos son similares a las de otros lenguajes: no es posible declarar dos veces un identificador en el mismo ámbito (pero sí en ámbitos diferentes), y en cuanto un ámbito se cierra se deben olvidar las variables declaradas en dicho ámbito.

7. Puesto que en la traducción las variables locales a los subprogramas se convierten en variables globales con un prefijo que depende del subprograma en que fue declarada la variable, y puesto que es posible que la variable se use en otros subprogramas (descendientes del que la declaró), una de las formas más sencillas de generar el prefijo correcto para una variable es almacenar en la tabla de símbolos la traducción completa de la variable, con prefijo y nombre de la variable. Seguramente existen otras soluciones, pero probablemente son más complicadas de implementar.

8. En la traducción de las expresiones, cuando se operan números reales y números enteros se deben generar las conversiones de tipo necesarias y utilizar los operadores específicos para cada tipo de dato, como se muestra en el ejemplo.

9. En el lenguaje fuente, los operadores relacionales producen un valor de tipo booleano que no puede operarse con ningún otro operando, ni asignarse a ninguna variable, ni imprimirse. Sin embargo, las expresiones de las instrucciones "`si`" y "`mientras`" deben ser de tipo booleano.

10. En el lenguaje fuente no es posible asignar una expresión de tipo real a una variable de tipo entero. Tampoco es posible asignar un valor booleano a una variable (sea entera o real).

11. La traducción de la instrucción "`escribir`" depende del tipo de la expresión, se debe generar la cadena "`%d`" si es entera o "`%g`" si es real.

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 instrucción de asignación 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 en el lenguaje fuente. Si corresponden a funciones, 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:

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

5. No se puede asignar a una variable real o entera una expresión relacional (booleana) :

```
Error semantico (fila,columna): ':=' no admite expresiones booleanas
```

6. No se pueden imprimir expresiones relacionales (booleanas):

```
Error semantico (fila,columna): 'escribir' no admite expresiones booleanas
```

7. Las expresiones de las instrucciones `si` y `mientras` deben ser de tipo relacional (booleano):

```
Error semantico (fila,columna): en la instrucción 'lexema' la expresión debe ser relacional
```

8. El operador `//` sólo se puede utilizar cuando ambos operandos son de tipo entero:

```
Error semantico (fila,columna): los dos operandos de '//' deben ser enteros
```

En el Moodle de la asignatura se publicará un fragmento de código con constantes y un método para emitir errores semánticos.

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 sintaxis del lenguaje fuente obliga a declarar las variables antes de conocer su tipo, por lo que deben declararse con cualquier tipo (p.ej. un -1) y guardarse en una lista para, una vez conocido el tipo, recorrer dicha lista buscando las variables en el ámbito actual asignándoles a los símbolos el tipo correcto.

4. 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. En algunos casos, los métodos pueden devolver la traducción con un "`String`", pero en muchos casos deben devolver más de un atributo sintetizado (p.ej. el tipo y la traducción); en esos casos será necesario utilizar otra clase con los atributos que tiene que devolver, de manera que el método asociado al no terminal devuelva un objeto de esta otra clase (es posible que sea necesario usar más clases para devolver distinta combinaciones de atributos sintetizados).
- 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, rea o función) 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
   ```