PROCESADORES DE LENGUAJES I
PRÁCTICA DE LABORATORIO 4

En esta práctica trabajaremos con ANTLR a nivel semántico utilizando gramáticas con atributos. ANTLR permite implementar con facilidad los dos modelos de evaluación de gramáticas con atributos: al vuelo y sobre árboles. En esta sesión nos encargaremos del primer modo, dejando para prácticas posteriores la evaluación sobre árboles.

Un evaluador simple

Utilizaremos como lenguaje ejemplo el de las expresiones aritméticas. En este caso además de reconocer el lenguaje nos interesa evaluar las distintas expresiones, por lo que para la entrada:

```
1+1-3;
2+5+(8-2);
1;
```

La salida sería del tipo:

```
Expresion: -1
Expresion: 13
Expresion: 1
```

Con lo ya visto en prácticas anteriores, el reconocimiento de este lenguaje queda resuelto con la siguiente especificación:

```
///////////////////////////////
// Analizador sintáctico
///////////////////////////////
class Anasint extends Parser;
entrada : (instruccion)* EOF;
instruccion : expr ";" ;
expr      : exp_mult
           (("+" exp_mult)
           |("-" exp_mult))* ;
exp_mult : exp_base
           (("*" exp_base)
           |("/" exp_base))* ;
exp_base : NUMERO
          | "(" expr ")"
          ;
```

La gramática es muy parecida a las utilizadas en prácticas anteriores para el lenguaje de las expresiones salvo que en esta ocasión las reglas que definen `exp_mul` y `exp_base` se han escrito de una forma menos compacta. Esto se ha hecho para separar las parejas de operadores `"+"|"-"` y `"*"|"/"` para poderlos atribuir por separado.

En las siguientes secciones introduciremos los elementos necesarios para ampliar este reconocedor de manera que además de determinar si una entrada es correcta o no sea capaz de evaluar las expresiones reconocidas.

El modelo de ejecución de ANTLR

Una gramática con atributos no es más que una gramática en la que se pueden asociar atributos a los símbolos y en la que se pueden incrustar acciones semánticas (código en el lenguaje de programación destino, en nuestro caso java). Para implementar gramáticas con atributos ANTLR se apoya en las facilidades que le proporciona su estrategia de implementación de reconocedores, en concreto:

- La representación de *tokens* a través de objetos, y
- la implementación de un reconocedor recursivo descendente.

Gracias a que los *tokens* son objetos resulta bastante cómodo asociarle y recuperar información de ellos. Por defecto ANTLR ya implementa el lexema, la fila y la columna, y ya vimos en la práctica dedicada al análisis léxico cómo añadir más atributos a los *tokens*. El analizador léxico será el encargado de crear los *tokens* y de actualizar sus atributos, de manera que sólo nos queda por saber cómo recuperar los *tokens* desde el analizador sintáctico. Para ello haremos uso de las etiquetas, que son identificadores que nos permiten dar un nombre a los *tokens* que aparecen en las reglas. Por ejemplo:

```
expresion : n:NUMERO
             {System.out.println(n.getText());}
          ;
```

En este caso n es la etiqueta que nos da acceso al *token* `NUMERO`, el uso de estas etiquetas es necesario ya que en algunas ocasiones podemos encontrar más de una vez el mismo token en la parte derecha de una regla:

```
comparacion : n1:NUMERO ">" n2:NUMERO
              {System.out.println(n1.getText()+">"+
                                  n2.getText());}
            ;
```

Por su parte la implementación de los reconocedores recursivos facilita bastante la traducción de las reglas que llevan incrustadas acciones semánticas. Con este esquema se implementa un método por cada símbolo no terminal, que será encargado del reconocimiento y en el que se incluirán en el lugar apropiado las acciones semánticas. Por ejemplo para la primera de las dos reglas anteriores se generará un método con la siguiente estructura[1]:

```
public final void expresion() throws
   RecognitionException, TokenStreamException {
   // Declaración del acceso al token NUMERO
   Token n = NULL;

   // Código correspondiente al reconocimiento
   //   de expresion: NUMERO
     ...
   // Acción semántica
   System.out.println(n.getText());
}
```

Como se puede observar la implementación del acceso a un *token* y la acción semántica son triviales. Para el *token* basta con utilizar la etiqueta para declarar una variable de la clase `Token`, mientras que la acción semántica se copia literalmente en el sitio apropiado.

[1] Es una simplificación pero la idea es la misma. Se puede ver la implementación completa de una regla como ésta abriendo el fichero java generado por ANTLR a partir de la gramática.

En los siguientes apartados veremos cómo implementar sobre este esquema de reconocimiento los flujos de información que suponen los atributos heredados y sintetizados.

Atributos sintetizados

Los atributos sintetizados se especifican en ANTLR a través de valores asociados al reconocimiento de una regla. Cuando estas reglas son traducidas a métodos java, estos atributos son implementados como valores de retorno de dichos métodos. La sintaxis utilizada es muy intuitiva:

```
exp_suma returns [int res=0] {int e1,e2;}
        : e1=exp_mult "+" e2=exp_mult
             {res=e1+e2;};
```

Antes de los dos puntos que dan comienzo a la parte derecha de la regla nos encontramos con dos declaraciones:

- `returns [int res=0]` que establece el nombre y el tipo del atributo que se sintetizará tras reconocer el símbolo `exp_suma`.
- `{int e1,e2;}` donde se declaran dos atributos que podrán ser utilizados en la parte derecha de la regla para almacenar resultados intermedios.

En la parte derecha de la regla aparecen ejemplos de cómo recuperar y actualizar el valor de atributos sintetizados:

- Con instrucciones del tipo `e1=exp_mult` indicamos que tras el reconocimiento del símbolo no terminal `exp_mult`, el atributo que sintetiza debe ser guardado en la variable `e1` (declarada previamente).
- La acción semántica `{res=e1+e2;}` establece cómo calcular el valor del atributo sintetizado `res`.

La traducción a java de esta regla es casi inmediata ya que prácticamente todos los elementos utilizados en la especificación respetan la sintaxis java:

```
public final int  exp_suma() throws
   RecognitionException, TokenStreamException {
   // Declaración del atributo sintetizado
   int res=0;
   // Declaración de las varaibles locales a
   //  la regla
   int e1,e2;

   // Código propio del reconocimiento
   ...

   // Reconocimiento de exp_mult y captura
   //  del atributo e1
   e1=exp_mult();

   // Reconocimiento de exp_mult y captura
   //  del atributo e2
   e2=exp_mult();

   // Acción semántica
   res=e1+e2;
```

```
        // Devolución del atributo sintetizado
        return res;
     }
```

Atributos heredados

De la misma forma que los atributos sintetizados se implementan a través de valores devueltos por métodos, los atributos heredados también se implementan con facilidad gracias al modelo de reconocimiento recursivo. En este caso utilizaremos los parámetros de los métodos asociados a los símbolos para transmitir la información en sentido descendente. Teniendo en mente este modelo de implementación, la sintaxis se explica por sí sola. Por ejemplo en la siguiente regla el símbolo `instruccion` hereda del símbolo `instrucciones` el atributo `nbw`, que indica el número de bucles `while` abiertos hasta el momento:

```
     instrucciones [int nbw]: (instruccion[nbw])*;
```

El correspondiente método de reconocimiento seguiría el siguiente esquema:

```
     public final void instrucciones(int nbw) throws
        RecognitionException, TokenStreamException {
        // Código propio del reconocimiento
        ...
        // Reconocimiento de instruccion y transmisión
        //  del atributo nbw
        instruccion(nbw);
     }
```

El atributo `nwb` que es heredado tanto por `instruccion` como por `instrucciones` es implementado como un parámetro de los métodos correspondientes. En este caso la transmisión consiste simplemente en propagar el valor recibido por `instrucciones` hacia el símbolo `instruccion`.

La especificación del evaluador

Dado que el analizador léxico actualiza por defecto el lexema de los *tokens*, tenemos ya de partida la información que necesitamos. De manera que los únicos cambios necesarios para transformar el reconocedor en evaluador habrá que hacerlos en el analizador sintáctico:

```
     ///////////////////////////////
     // Analizador sintáctico
     ///////////////////////////////
     class Anasint extends Parser;

     entrada : (instruccion)* EOF;
     instruccion {int e;}
               : e=expr ";"
                   {System.out.println("Expresion: "+e);}
               ;
     expr returns [int res=0] {int e1,e2;}
               : e1=exp_mult {res=e1;}
                 (("+" e2=exp_mult {res=res+e2;})
                 |("-" e2=exp_mult {res=res-e2;}))*
               ;
```

```
exp_mult returns [int res=0] {int e1,e2;}
          : e1=exp_base {res=e1;}
            (("*" e2=exp_base {res=res*e2;})
            |("/" e2=exp_base {res=res/e2;}))*
          ;
exp_base returns [int res=0] {int e;}
          : n:NUMERO
           {res = new
                  Integer(n.getText()).intValue();}
          | "(" e=expr ")"
           {res = e;}
          ;
```

En este caso se han utilizado sólo atributos sintetizados. De esta forma se consigue combinar la información proporcionada por los símbolos terminales (números y operadores) para calcular los valores de las diferentes expresiones.

Atributos propios del analizador

En algunas ocasiones puede ser interesante disponer de atributos que sean accesibles desde cualquier punto del analizador. Por ejemplo, cuando se pretenda ampliar el lenguaje de la calculadora para que incluya variables será muy útil disponer de una tabla *hash* en la que registrar los valores que van tomando las variables en cada momento. Este tipo de atributos pueden ser declarados al principio del analizador justo antes de la primera regla. Para el caso de la tabla de variables podríamos hacer algo así:

```
header{
import java.util.Hashtable;
}
///////////////////////////////////
// Anasint.g: Analizador sintáctico
///////////////////////////////////
class Anasint extends Parser;
{Hashtable variables = new Hashtable();}

instrucciones: (instruccion) + ;
...
```

En este caso, además, se ha hecho uso de la sección `header` para importar la clase `Hashtable` del paquete `java.util`.

Ejercicios

1. Compilar la especificación del evaluador presentada en el enunciado y comprobar su funcionamiento.
2. Ampliar el lenguaje del apartado 1 de manera que incluya variables e instrucciones de asignación. La entrada será una secuencia de expresiones y asignaciones, por ejemplo:

   ```
   a := 10+1;
   a*2;
   b:=a*10;
   ```

   La salida en este caso consistirá en una línea por cada instrucción procesada, indicando si la instrucción es una asignación o una expresión. En el caso de las expresiones sólo se informará del valor, mientras que para las asignaciones se indicará también la variable que ha sido modificada. Para el ejemplo anterior la salida esperada sería por tanto:

```
Asignación (a) => 11
Expresión => 22
Asignación (b) => 110
```

Una variable que no haya sido asignada tendrá asociado inicialmente el valor 0.

3. Especificar un reconocedor que use atributos heredados para determinar si la instrucción `break` de java está ubicada, o no, dentro de algún bucle `while`. Por ejemplo en el siguiente ejemplo la respuesta sería no:

```
if (a) {
   break;
}
```

mientras que para el siguiente la respuesta sería sí:

```
while(a) {
   if (a) {
      break;
   }
}
```

4. Tomando como base el resultado del ejercicio 3 de la práctica 3, especificar un reconocedor que produzca como salida una versión *formateada* del programa en la que se respetan ciertas normas de estilo. Por ejemplo para un fragmento de programa como el que sigue:

```
void main(void) {int a, b;if
(2*a>=b) {printf
("punto medio %d\n",a);a=a+1;}}
```

La versión *formateada* será:

```
void main(void) {
   int a, b;

   if(2*a>=b) {
      printf("punto medio %d\n",a);
      a=a+1;
   }
}
```

5. Atribuir la siguiente gramática simplificada de XML para que se compruebe que los nombres de las correspondientes etiquetas de apertura y cierre coinciden:

```
elementos : (elemento)*
          ;
elemento : APERTURA elementos CIERRE
         | TEXTO
         ;
```

¿Qué utilidad tendría en este problema un predicado semántico?

6. Capturar la excepción generada al intentar ejecutar una división por cero (`ArithmeticException`). ANTLR dispone de la palabra reservada `exception` para tal fin. En el siguiente ejemplo se muestran los distintos puntos dentro de una regla en los que se pueden capturar excepciones:

```
regla:   a:A B C A
     |   D E
         exception // para la alternativa "D E"
         catch [Exception ex] {
            ...}
     ;
```

```
exception // para la regla completa
catch [Exception ex] {
   ...}
exception[a] // primera aparición de "A"
             // etiquetada con "a"
catch [Exception ex] {
   ...}
```