Compiladores
PL-3
"Visitors en ANTLR"

Katia Leal Algara

Notas:

- Visitors en ANTLR
- Generación de código intermedio con Jasmin
- Análisis semántico
- Gestión de errores

Gramática de ejemplo: Demo.g4

```
grammar Demo;

addition: left=addition '+' right=NUMBER #Plus
          | number=NUMBER #Number
          ;

NUMBER: [0-9]+;
```

Sólo tiene una regla, *addition*, lo realmente interesante de esta gramática es que se le han añadido **etiquetas** para poder identificar puntos de la gramática en los que queremos realizar alguna acción, ya sea para generar pseudocódigo, para realizar comprobaciones de tipo semántico o gestión de errores.

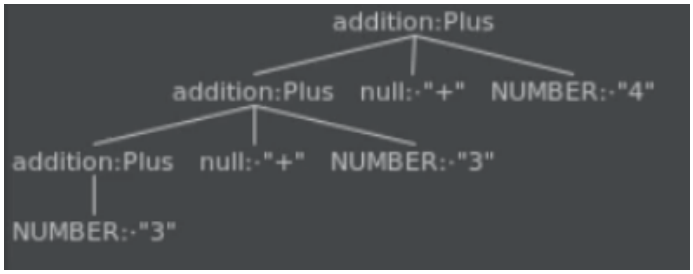
Podemos hacer referencia tanto a la regla completa como a partes de la misma, a todos o algunos de sus elementos. Para poder diferenciar las distintas etiquetas, las etiquetas para hacer referencia a una regla comienzan con una mayúscula, como `Plus` y `Number`, mientras que las que hacen referencia a elementos de una regla están escritas en minúsculas, como `left`, `right` y `number`.

Compilar gramática con ANTLR

```
$ antlr4 -no-listener -visitor Demo.g4
```

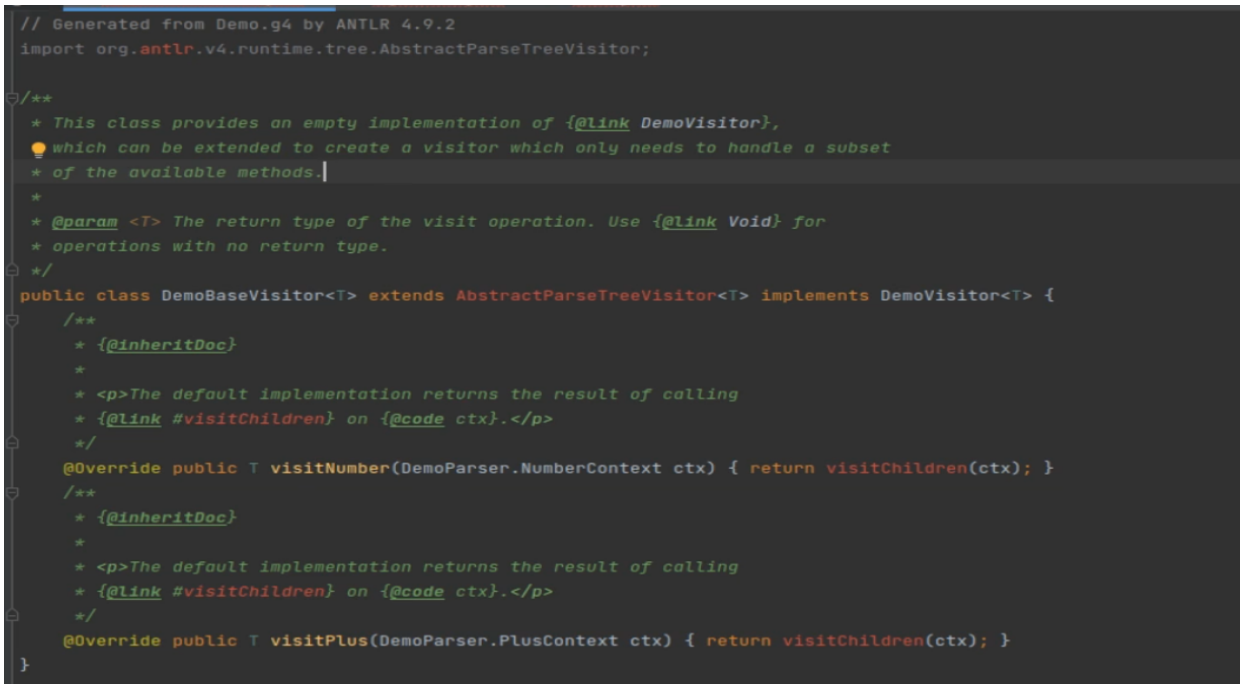
Se puede elegir si generar *listeners* y *visitors* o hacer todo con los *visitors*.
Una vez compilado, ha generado la clase DemoBaseVisitor.java.

Si, por ejemplo, pasamos la entrada 3+3+4, el árbol resultante sería como el siguiente:



Clase DemoBaseVisitor.java: MyVisitor.java

Una vez compilada la gramática con ANTLR, entre otras, se crea la clase **DemoBaseVisitor.java**:



```java
// Generated from Demo.g4 by ANTLR 4.9.2
import org.antlr.v4.runtime.tree.AbstractParseTreeVisitor;

/**
 * This class provides an empty implementation of {@link DemoVisitor},
 * which can be extended to create a visitor which only needs to handle a subset
 * of the available methods.
 *
 * @param <T> The return type of the visit operation. Use {@link Void} for
 * operations with no return type.
 */
public class DemoBaseVisitor<T> extends AbstractParseTreeVisitor<T> implements DemoVisitor<T> {
    /**
     * {@inheritDoc}
     *
     * <p>The default implementation returns the result of calling
     * {@link #visitChildren} on {@code ctx}.</p>
     */
    @Override public T visitNumber(DemoParser.NumberContext ctx) { return visitChildren(ctx); }
    /**
     * {@inheritDoc}
     *
     * <p>The default implementation returns the result of calling
     * {@link #visitChildren} on {@code ctx}.</p>
     */
    @Override public T visitPlus(DemoParser.PlusContext ctx) { return visitChildren(ctx); }
}
```

Esta clase incluye los métodos *visitNumber* y *visitPlus* que se corresponden con las etiquetas que hemos puesto a las dos reglas de la gramática de ejemplo.

Vamos a extender esta clase para crear nuestro propio visitor, **MyVisitor.java**.

```java
public class MyVisitor extends DemoBaseVisitor<String> {

    @Override
    public String visitPlus(DemoParser.PlusContext ctx) {
        return visitChildren(ctx) + "\nldc " + ctx.right.getText() + "\n" + "iadd";
    }

    @Override
    public String visitNumber(DemoParser.NumberContext ctx) {
        return "ldc " + ctx.number.getText();
    }

    @Override
    protected String aggregateResult(String aggregate, String nextResult) {
        if (aggregate == null) {
            return nextResult;
        }
        if (nextResult == null) {
            return aggregate;
        }
        return aggregate + "\n" + nextResult;
    }

}
```

Cuando visito un “+”, tengo que visitar los hijos por la parte izquierda: `visitChildren(ctx)`. Es en estos nodos en los que se tienen que ir añadiendo las instrucciones Jasmin para generar el código ensamblador. Igualmente, podemos usar los visitors para implementar un analizar semántico y para la gestión de errores.

Programa principal: Main.java

```
public class Main {

    public static void main(String[] args) throws Exception {
        String inputFile = null;

        if (args.length > 0) {
            inputFile = args[0];
        }
        CharStream inputstream = null;
        if (inputFile != null) {
            System.out.println(compile(inputFile));
            }
    }

    public static String compile(String inputFile) throws IOException {
        DemoLexer lexer = new DemoLexer(CharStreams.fromFileName(inputFile));
        DemoParser parser = new DemoParser(new CommonTokenStream(lexer));

        ParseTree tree = parser.addition();
        //System.out.print(new MyVisitor().visit(tree));
        return createJasminFile(new MyVisitor().visit(tree));
    }

    private static String createJasminFile(String instructions) {
        return ".class public Sumar\n"
                + ".super java/lang/Object\n"
                + "\n"
                + ".method public static main([Ljava/lang/String;)V\n"
                + "   .limit stack 100\n"
                + "   .limit locals 100\n"
                + "\n"
                + "   getstatic java/lang/System/out Ljava/io/PrintStream;\n"
                + instructions + "\n"
                + "   invokevirtual java/io/PrintStream/println(I)V\n"
                + "return\n"
                + "\n"
                + ".end method";
    }
}
```

Para compilar el Main:

```
$ javac *.java
```

Para ejecutar Main con el fichero code.demo como entrada:

```
$ java Main code.demo
```

Ejemplo: code.demo y Sumar.j

Si a la entrada de nuestro programa/compilador introducimos 3+3+4+5+10, nos devolverá como salida el contenido del fichero Sumar.j (podemos hacer una redicción de la salida para volcarla al fichero elegido).

```
.class public Sumar
.super java/lang/Object

.method public static main([Ljava/lang/String;)V
   .limit stack 100
   .limit locals 100

   getstatic java/lang/System/out Ljava/io/PrintStream;
ldc 3
ldc 3
iadd
ldc 4
iadd
```

```
ldc 5
iadd
ldc 10
iadd
   invokevirtual java/io/PrintStream/println(I)V
return

.end method
```

Si a la entrada de nuestro programa/compilador introducimos 3+3+4+5+10, nos devolverá como salida el contenido del fichero Sumar.j (podemos hacer una redicción de la salida para volcarla al fichero elegido).

Ensamblar un fichero Jasmin

Para ensamblar un fichero Jasmin hay que estar en el directorio en el que hemos descargado Jasmin para ejecutar el .jar:

```
$ java -jar jasmin.jar Sumar.j
```

Después de ejecutar Jasmin, se crea el fichero compilado Sumar.class. Sólo resta ejecutar Sumar de la siguiente manera:

```
$ java Sumar
25
```

Otros ejemplos Jasmin

Vacio.j

```
.class public Vacio
.super java/lang/Object

.method public static main([Ljava/lang/String;)V

   ; reservar espacio en la pila, podria ser 0 aqui
   .limit stack 0

   ; terminar el main
   return

.end method
```

HolaMundo.j

```
.class public HolaMundo
.super java/lang/Object

.method public static main([Ljava/lang/String;)V

   ; reservamos 2 sitios en la pila
   .limit stack 2

   ; ponemos a java.lang.System.out (type PrintStream) en la pila
   getstatic java/lang/System/out Ljava/io/PrintStream;

   ; hacemos un push en la pila de la cadena de caracteres deseada
   ldc "Hola Mundo!"

   ; invocamos la funcion println
   invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

   ; y cerramos el main
   return

.end method
```