ANTLR (ANother Tool for Language Recognition, aunque algunos aseguran con cierta malicia que el acrónimo corresponde a ANTi-LR) es una herramienta de código abierto escrita en Java que permite construir semiautomáticamente traductores entre lenguajes informáticos. A partir de una gramática y de las correspondientes acciones semánticas, ANTLR genera un analizador sintáctico descendente recursivo que va ejecutando dichas acciones conforme analiza el fichero de entrada. El analizador sintáctico utiliza el algoritmo LL(*), una generalización del LL(k); para evitar algunos conflictos producidos en el algoritmo LL(*) a la hora de analizar sintácticamente ciertas clases de gramáticas, ANTLR permite añadir predicados sintácticos o semánticos e, incluso, realizar retroceso (*backtracking*) durante el análisis. ANTLR permite escribir las acciones semánticas en varios lenguajes: Java, C#, C++, Python, C, etc. Aquí nos centraremos en Java.

Un ejemplo sencillo

Comencemos con un sencillo ejemplo de especificación de un traductor de expresiones aritméticas infijas a notación prefija; la especificación de este traductor puede encontrarse en Prefija.g (los ficheros de ANTLR suelen tener la extensión .g). El fichero Prefija.g contiene, en este caso, tanto la especificación del analizador léxico como la del sintáctico; ANTLR permite también indicarlas en ficheros separados.

Todos los ficheros de ANTLR comienzan con la palabra grammar seguida del nombre del fichero (sin extensión). A continuación, aparecen diversas secciones de código marcadas con un nombre que identifica dónde se situarán en el código generado por ANTLR:

- header: el código que aparece antes de la definición de la clase; habitualmente se colocan aquí las directivas para importar o definir paquetes
- members: contiene variables o métodos que formarán parte de la clase
- rulecatch: permite sobrescribir las cláusulas catch que genera ANTLR para capturar los errores

ANTLR genera para esta especificación dos clases diferentes, como ya se ha comentado: una para el analizador léxico y otra para el sintáctico; estas clases son, respectivamente, PrefijaLexer y PrefijaParser. Las secciones de código anteriores se situarán en la clase PrefijaParser, salvo que se precedan de la palabra lexer:

```
@members {
    /* Miembros de la clase del analizador sintáctico */
}

@lexer::members {
    /* Miembros de la clase del analizador léxico */
}
```

En nuestro ejemplo, las reglas del analizador sintáctico usan el tipo String de Java, por lo que es necesario añadir la directiva import correspondiente. La sección rulecatch evita que el analizador sintáctico generado por ANTLR se detenga en el primer error y no realice recuperación de errores.

A continuación aparecen las reglas del analizador sintáctico junto con las acciones semánticas correspondientes. Las reglas de la gramática se expresan en notación EBNF. Cada regla finaliza con un punto y coma, y pueden especificarse varias reglas con la misma parte izquierda si las partes derechas se separan una barra. Los no terminales se escriben en minúsculas y los tokens en mayúsculas. Así la especificación de ANTLR:

```
atom
    :   INT
    |   ID
    |   '(' expr ')'
    ;
```

corresponde con la gramática:

Atom → **int**
Atom → **id**
Atom → **(** Expr **)**

Cuando un token corresponde a un patrón específico (como los paréntesis de este ejemplo), suele ser más cómodo introducir el lexema entre comillas simples en lugar de darle un nombre y definirlo en la sección del analizador léxico; desde el punto de vista funcional, ambos métodos son

completamente equivalentes. En el caso de que el patrón corresponda a más de un carácter también se utilizarán las comillas simples. Es decir, en la especificación de ANTLR tanto los caracteres sencillos como las cadenas se encierran entre comillas simples; una cosa bien distinta es en el cuerpo de las acciones semánticas: allí es necesario seguir las reglas del lenguaje objetivo empleado.

Los atributos sintetizados de un ETDS se transforman en valores de retorno de los métodos asociados a cada no terminal cuando el ETDS se implementa sobre un analizador descendente recursivo. Supongamos un no terminal A que tiene asociados dos atributos sintetizados, i y s, de tipo entero y cadena, respectivamente; en la especificación de ANTLR esta circunstancia se expresa con:

```
a returns [int i, String s]
   : ...
   ;
```

Estos atributos se manejan como cualquier otra variable, con la excepción de que su nombre debe ir precedido del símbolo $ (es decir, $i y $s), como puedes ver en las reglas de la especificación de ANTLR. Allí también puedes ver cómo se recogen los atributos sintetizados devueltos por un no terminal:

```
c ...
   : r=a {...acceso a $r.i y $r.s...}
   ;
```

ANTLR también permite una notación del tipo $a.i y $a.r, con lo que nos evitamos el uso de variables como r en el ejemplo anterior; no obstante, esto solo funciona cuando no hay otro no terminal con el mismo nombre ni en la parte izquierda ni en la derecha de la regla actual.

Las variables locales a una regla (y, en general, cualquier código que deba aparecer antes del código generado para la regla) se declaran mediante una acción especial denominada init:

```
a returns [int i, String s]
@init{
   int variableLocal;
}
   : ...acceso a variableLocal...
   ;
```

Los atributos de un *token* se pueden acceder directamente como se ve en las reglas del no terminal Atom: así, $ID.text es el lexema del *token* ID. Este atributo es inicializado directamente por el analizador léxico, como también lo son los atributos line y pos, que contienen, respectivamente, la fila (empezando desde 1) y columna (empezando desde 0) del *token*. En el caso de que haya más de un *token* con el mismo nombre en una misma regla, es necesario recurrir a la misma notación que en el caso de los no terminales:

```
f : id1=ID  '('  id2=ID  ')'  {...acceso a $id1.text e $id2.text...}
  ;
```

Después de la especificación sintáctica, aparece la especificación léxica. Como puedes ver, cada token va seguido de una expresión regular que lo define; además, es posible añadir acciones semánticas, aunque esto suele ser necesario solo en casos especiales. Uno de estos casos es el del tratamiento de los blancos: ANTLR ignora el *token* detectado y busca el siguiente *token* cuando la acción semántica incluye una llamada a skip(). Los comentarios también se suelen ignorar con este esquema, como puedes ver en la especificación léxica del ejemplo.

Si una entrada particular concuerda con el patrón regular de más de un *token*, el analizador léxico de ANTLR devuelve el que aparezca primero en la especificación; por esta razón, es importante en particular situar las las palabras reservadas antes del *token* correspondiente al identificador.

Para poder lanzar el traductor generado por ANTLR es necesario escribir un programa principal que invoque los métodos adecuados. Este código no suele ser muy distinto de un proyecto a otro; la clase Main.java es adecuada para ejecutar nuestro ejemplo de traducción a notación prefija.

Para probar el ejemplo, puedes utilizar las siguientes órdenes desde la línea de comandos para compilar y ejecutar el ejemplo, suponiendo que tienes el fichero antlr-3.4-complete.jar descargado:

- Para compilar el ejemplo, habría que poner:

```
java -classpath antlr-3.4-complete.jar  org.antlr.Tool   Prefija.g
javac -classpath antlr-3.4-complete.jar:. Main.java PrefijaLexer.java PrefijaParser.java
```

- Para ejecutar el código generado por ANTLR, habría que poner:

```
java -classpath antlr-3.4-complete.jar:. Main   <fichero>
```

Hazlo tú ahora. Estudia el código de las clases generadas por ANTLR e intenta deducir cómo funciona.

Otro ejemplo con atributos heredados

Muchos ETDS utilizan atributos heredados para propagar información por el árbol de análisis sintáctico. En un analizador descendente los atributos heredados se transforman en parámetros de los métodos correspondientes y ANTLR utiliza una notación similar para expresarlos. La especificación de ANTLR Prefija.g y el programa principal Main.java implementan la versión en BNF de la traducción a notación prefija, lo que implica reescribir las recursividades por la izquierda de la gramática de expresiones original; esta transformación, a su vez, sugiere el uso de atributos heredados en el ETDS.

Supongamos un no terminal A que tiene asociados dos atributos heredados, h y t, de tipo entero y cadena, respectivamente; en la especificación de ANTLR esta circunstancia se expresa con:

```
a [int h, String t] returns [int i, String s]
   : ...
   ;
```

Estos atributos se manejan como cualquier otra variable, con la excepción de que su nombre debe ir precedido del símbolo $ (es decir, $h y $t), como puedes ver en las reglas de la especificación de ANTLR. Allí también puedes ver cómo se pasan los atributos heredados a un no terminal:

```
c ...
   : r=a[22,"real"] ...
   ;
```

Para acabar un par de comentarios finales: ANTLR permite construir árboles sintácticos abstractos (*abstract syntax trees*, AST) como paso intermedio en el proceso de traducción; aquí, sin embargo, nos centraremos en acciones semánticas que construyen directamente la traducción final conforme se ejecuta el algoritmo de análisis sintáctico. Por otro lado, ANTLR utiliza para el analizador léxico generado el mismo algoritmo que el usado en el analizador sintáctico. Esto aumenta la potencia del analizador léxico, pero aquí no entraremos en detalles al respecto; si en la especificación léxica se acompaña cada *token* de una expresión regular independiente, el comportamiento del algoritmo es básicamente equivalente al estudiado en clase de teoría.

¿Por qué ANTLR?

- Su desarrollo tiene un gran nivel de actividad.
- Aunque Coco/R es probablemente un poco más didáctico y rápido, su uso no está tan extendido. Sin embargo, una ventaja de Coco/R es que no necesita una librería externa para poder ejecutar los traductores resultantes.
- Al generar analizadores sintácticos descendentes (al contrario que Bison o yacc), los atributos heredados y sintetizados del ETDS se mantienen sin apenas modificaciones en la implementación.
- En un analizador descendente, la inclusión de acciones semánticas nunca puede introducir nuevos conflictos, al contrario de lo que ocurre con Bison o yacc.
- Permite usar varios lenguajes como lenguajes objetivo; entre ellos, C#, Java, C++, C o Python.
- En las últimas versiones utiliza el algoritmo de análisis sintáctico LL(*) que evita muchos de los conflictos que muchas gramáticas producen con los analizadores LL(k).
- Existen herramientas auxiliares como la extensiones de Eclipse o la interfaz gráfica ANTLRWorks.
- Dispone de una gran cantidad de gramáticas.

Para saber más

- The Definitive ANTLR Reference
- Five Minute Introduction to ANTLR 3
- Una entrevista con el autor de ANTLR
- Otra entrevista con el autor
- Una guía muy completa en castellano.