Capítulo 5

JavaCC

5.1 Introducción

JavaCC (*Java Compiler Compiler* - Metacompilador en Java) es el principal metacompilador en JavaCC, tanto por sus posibilidades, como por su ámbito de difusión. Se trata de una herramienta que facilita la construcción de analizadores léxicos y sintácticos por el método de las funciones recursivas, aunque permite una notación relajada muy parecida a la BNF. De esta manera, los analizadores generados utilizan la técnica descendente a la hora de obtener el árbol sintáctico.

5.1.1 Características generales

JavaCC integra en una misma herramienta al analizador lexicográfico y al sintáctico, y el código que genera es independiente de cualquier biblioteca externa, lo que le confiere una interesante propiedad de independencia respecto al entorno. A grandes rasgos, sus principales características son las siguientes:

- Genera analizadores descendentes, permitiendo el uso de gramáticas de propósito general y la la utilización de atributos tanto sintetizados como heredados durante la construcción del árbol sintáctico.
- Las especificaciones léxicas y sintácticas se ubican en un solo archivo. De esta manera la gramática puede ser leída y mantenida más fácilmente. No obstante, cuando se introducen acciones semánticas, recomendamos el uso de ciertos comentarios para mejorar la legibilidad.
- Admite el uso de estados léxicos y la capacidad de agregar acciones léxicas incluyendo un bloque de código Java tras el identificador de un token.
- Incorpora distintos tipos de tokens: normales (TOKEN), especiales (SPECIAL_TOKEN), espaciadores (SKIP) y de continuación (MORE). Ello permite trabajar con especificaciones más claras, a la vez que permite una mejor gestión de los mensajes de error y advertencia por parte de JavaCC en tiempo de metacompilación.
- Los tokens especiales son ignorados por el analizador generado, pero están disponibles para poder ser procesados por el desarrollador.
- La especificación léxica puede definir tokens de manera tal que no se diferencien las mayúsculas de las minúsculas bien a nivel global, bien en un patrón concreto.
- Adopta una notación BNF propia mediante la utilización de símbolos propios

de expresiones regulares, tales como (A)*, (A)+.

- Genera por defecto un analizador sintáctico LL(1). No obstante, puede haber porciones de la gramática que no sean LL(1), lo que es resuelto en JavaCC mediante la posibilidad de resolver las ambigüedades desplazar/desplazar localmente en el punto del conflicto. En otras palabras, permite que el a.si. se transforme en LL(k) sólo en tales puntos, pero se conserva LL(1) en el resto de las reglas mejorando la eficiencia.
- De entre los generadores de analizadores sintácticos descendentes, JavaCC es uno de los que poseen mejor gestión de errores. Los analizadores generados por JavaCC son capaces de localizar exactamente la ubicación de los errores, proporcionando información diagnóstica completa.
- Permite entradas codificadas en Unicode, de forma que las especificaciones léxicas también pueden incluir cualquier carácter Unicode. Esto facilita la descripción de los elementos del lenguaje, tales como los identificadores Java que permiten ciertos caracteres Unicode que no son ASCII.
- Permite depurar tanto el analizador sintáctico generado como el lexicográfico, mediante las opciones DEBUG_PARSER, DEBUG_LOOKAHEAD, y DEBUG_TOKEN_MANAGER.
- JavaCC ofrece muchas opciones diferentes para personalizar su comportamiento y el comportamiento de los analizadores generados.
- Incluye la herramienta JJTree, un preprocesador para el desarrollo de árboles con características muy potentes.
- Incluye la herramienta JJDoc que convierte los archivos de la gramática en archivos de documentación.
- Es altamente eficiente, lo que lo hace apto para entornos profesionales y lo ha convertido en uno de los metacompiladores más extendidos (quizás el que más, por encima de JFlex/Cup).

JavaCC está siendo mantenido actualmente por el grupo java.net de código abierto, y la documentación oficial puede encontrarse en el sitio https://javacc.dev.java.net. Por desgracia, dicha documentación es demasiado formal y no incorpora ningún tutorial de aprendizaje, aunque constituye la mejor fuente de información una vez familiarizado ligeramente con la herramienta, lo que es el objetivo del presente apartado.

5.1.2 Ejemplo preliminar.

Al igual que con Cup, comenzaremos nuestra andadura con un ejemplo preliminar que dé idea de la estructura de un programa de entrada a JavaCC y de la notación utilizada para ello. Se usará el nombre «JavaCC» para hacer referencia tanto a la herramienta JavaCC como al lenguaje que ésta admite para describir gramáticas.

Así, nuestro ejemplo escrito en JavaCC es:

```
options {
        LOOKAHEAD=1;
}
PARSER_BEGIN(Ejemplo)
        public class Ejemplo{
            public static void main(String args[]) throws ParseException {
                Ejemplo miParser = new Ejemplo(System.in);
                miParser .listaExpr();
            }
        }
PARSER_END(Ejemplo)

SKIP :{
            " "
        |   "\t"
        |   "\n"
        |   "\r"
}

TOKEN [IGNORE_CASE] : {
            <ID: (["a"-"z"])+>
        |   <NUM: (["0"-"9"])+>
}

void listaExpr() : {} {
            ( exprBasica() ";")+
}
void exprBasica() :{}{ LOOKAHEAD(2)
            <ID> "(" expr() ")"
        |   "(" expr() ")"
        |   "@NEW" <ID>
        |   <ID> "." <ID>
}
void expr() :{}{
            "@ALGO"
        |   <NUM>
}
```

Los programas JavaCC se suelen almacenar en ficheros con extensión **.jj**. Así, el ejemplo anterior se almacenaría en el fichero **Ejemplo.jj**, de manera que la metacompilación se hace invocando al fichero por lotes **javacc.bat** (quien a su vez invoca a la clase **javacc.class** del fichero **javacc.jar**) de la forma:

```
javacc Ejemplo.jj
```

lo que producirá los ficheros de salida:

- Ejemplo.java: es el analizador sintáctico.
- EjemploTokenManager.java: es el analizador lexicográfico.
- EjemploConstants.java: interface que asocia un código a cada token.

Asimismo, almacena en forma de String el nombre dado por el desarrollador a cada token, con objeto de que el analizador sintáctico pueda emitir mensajes de error claros y explicativos.

además de las clases necesarias:

- ParseException.java: clase que implementa los errores sintácticos. Cuando el analizador sintáctico encuentra uno de estos errores genera una excepción que puede ser capturada en cualquier nodo ancestro del árbol sintáctico que se estaba construyendo.
- SimpleCharStream.java: clase que implementa los canales de entrada de los cuales puede leer el analizador léxico. Incorpora constructores para convertir los canales de tipo tradicional: **InputStream** y **Reader**.
- Token.java: clase que implementa el objeto a través del cual se comunican el analizador léxico y el sintáctico.
- TokenMgrError.java: clase que implementa los errores lexicográficos. Este tipo de errores no se puede capturar de forma fácil, por lo que el desarrollador debe recoger todos los lexemas posibles y generar sus propios errores ante los incorrectos.

si no existían ya en el directorio desde el que se metacompila. El proceso puede apreciarse en la figura 5.1. Para compilar todos los ficheros producidos por JavaCC basta compilar el fichero principal de salida (**Ejemplo.java** según la figura) quien produce la compilación en cadena de todos los demás.

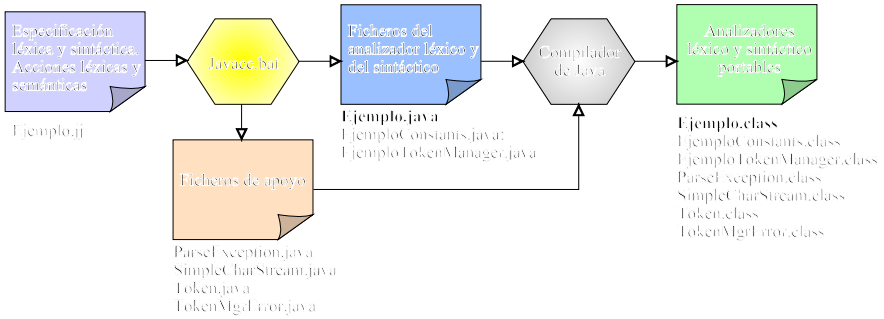

Figure 1 Proceso de metacompilación con JavaCC. Partiendo de un solo fichero (**Ejemplo.jj**), JavaCC produce 3 ficheros dependientes y otros 4 que son siempre idénticos.

5.2 Estructura de un programa en JavaCC

Como puede verse en el ejemplo propuesto, la estructura básica de un programa JavaCC es:

```
options {
    Área de opciones
}
```

PARSER_BEGIN(NombreClase)
    Unidad de compilación Java con la clase de nombre Nombreclase
PARSER_END(NombreClase)
Área de *tokens*
Área de funciones BNF

El área de opciones permite especificar algunas directrices que ayuden a JavaCC a generar analizadores léxico-sintácticos bien más eficientes, bien más adaptados a las necesidades concretas del desarrollador. En el ejemplo se ha indicado que, por defecto, la gramática indicada es de tipo LL(1), excepto si, en algún punto, se indica otra cosa.

Las cláusulas **PARSER_BEGIN** y **PARSER_END** sirven para indicarle a JavaCC el nombre de nuestra clase principal, así como para englobar tanto a ésta como a cualesquiera otras que se quieran incluir de apoyo, como pueda ser p.ej. un gestor de tablas de símbolos. En el ejemplo puede observarse que la clase principal constituye el analizador sintáctico en sí ya que la función **main()** crea un objeto de tipo **Ejemplo** a la vez que le pasa como parámetro en el constructor la fuente de la que se desea consumir la entrada: el teclado (**System.in**).

La clase creada por JavaCC incorporará una función por cada no terminal del área de reglas. Cada función se encargará de consumir la parte de la entrada que subyace debajo de su no terminal asociado en el árbol sintáctico de reconocimiento. Por tanto, asumiendo que el axioma inicial es **listaExpr**, una llamada de la forma **miParser.listaExpr()** consumirá toda la entrada, si ésta es aceptable.

Las siguientes dos áreas pueden mezclarse, aunque lo más usual suele ser indicar primero los *tokens* y finalmente las reglas en BNF, especialmente por motivos de claridad en el código.

En el ejemplo se han indicado *tokens* de dos tipos. Los *tokens* agrupados bajo la cláusula **SKIP** son aquellos que serán consumidos sin ser pasados al analizador sintáctico; en nuestro caso son: el espacio, el tabulador, el retorno de carro (CR-*Carry Return*) y la alimentación de línea (LF-*Line Feed*).

Los *tokens* bajo la cláusula **TOKEN** constituyen los *tokens* normales, aunque el desarrollador también puede indicar este tipo de *tokens* en las propias reglas BNF, como ocurre con los patrones "(", ")", ";", etc. La declaración de cada *token* se agrupa entre paréntesis angulares y está formada por el nombre del *token* seguido por el patrón asociado y separado de éste por dos puntos. Los patrones lexicográficos se describen de forma parecida a PCLex. El ejemplo ilustra el reconocimiento de un identificador formado sólo por letras (ya sean mayúsculas o minúsculas merced al modificador **[IGNORE_CASE]** de la cláusula **TOKEN**) y de un número entero.

La última área del ejemplo ilustra la creación de tres no terminales y sus reglas BNF asociadas. Dado que cada no terminal va a ser convertido por JavaCC en una función Java, su declaración es idéntica a la de dicha función y está sujeta a las mismas

restricciones que cualquier otra función en Java. El cuerpo de cada una de estas funciones será construido por JavaCC y tendrá como propósito el consumo adecuado de *tokens* en base a la expresión BNF que se indique en la especificación.

La notación BNF empleada hace uso del símbolo * que representa repetición 0 ó más veces, + para la repetición 1 ó más veces, | para la opcionalidad, paréntesis para agrupar, etc. Los terminales pueden indicarse de dos formas, bien colocando entre paréntesis angulares alguno de los declarados en la fase anterior, o bien indicando su patrón directamente si éste está formado por una cadena constante de caracteres (realmente, JavaCC permite cualquier expresión regular, pero ello no resulta útil para nuestros propósitos). En el ejemplo es de notar la inclusión del carácter "@" en los terminales *ad hoc* **NEW** y **ALGO**, ya que, en caso contrario serían considerados identificadores al encajar por el patrón del *token* **ID**. Los patrones "NEW" y "ALGO" habrían sido correctamente reconocidos si el *token* **ID** se hubiera declarado después de la aparición de éstos en las reglas BNF.

A continuación se estudiará con mayor detenimiento cada uno de estos apartados.

5.2.1 Opciones.

Esta sección comienza con la palabra reservada **options** seguida de una lista de una o más opciones entre llaves. La sección al completo puede omitirse, ya que no es obligatoria

Las opciones pueden especificarse tanto en el archivo de la gramática como en la línea de comandos. La misma opción no debe establecerse más de una vez, aunque en el caso de que se especifiquen en la línea de comandos, éstas tienen mayor prioridad que las especificadas en el archivo.

Los nombres de las opciones se escriben en mayúsculas, finalizan en ";" y se separan del valor que se les asocia mediante el signo "=". A continuación se detallan brevemente las opciones más importantes y las funciones que realizan:

LOOKAHEAD: indica la cantidad de *tokens* que son tenidos en cuenta por el analizador sintáctico antes de tomar una decisión. El valor por defecto es 1 lo que realizará análisis LL(1). Cuanto más pequeño sea el número de *tokens* de prebúsqueda, más veloz se realizarán los análisis. Este valor se puede modificar en determinadas reglas concretas.

CHOICE_AMBIGUITY_CHECK: sirve para que JavaCC proporcione mensajes de error más precisos cuando se encuentra con varias opciones con un prefijo de longitud igual o superior a la de **LOOKAHEAD**. En estos casos, JavaCC suele informar de que se ha encontrado un error y aconseja que se pruebe con un *lookahead* igual o superior a un cierto número **n**. Si **CHOICE_AMBIGUITY_CHECK** vale **k**, entonces JavaCC intentará averiguar el *lookahead* exacto que necesitamos (tan sólo a efectos de

informarnos de ello) siempre y cuando éste no supere el valor **k**. La utilización de esta opción proporciona información más precisa, pero a costa de un mayor tiempo de metacompilación. Su valor por defecto es 2.

FORCE_LA_CHECK: uno de los principales problemas que puede encontrarse JavaCC en la construcción del analizador sintáctico radica en la existencia de prefijos comunes en distintas opciones de una misma regla, lo que hace variar el número de *tokens* de prebúsqueda necesarios para poder escoger una de estas opciones. Cuando la opción **LOOKAHEAD** toma el valor 1, JavaCC realiza un control de errores mediante chequeos de prefijo en todas las opciones en las que el desarrollador no haya indicado un **LOOKEAHEAD** local, informando por pantalla de tales situaciones. Si **LOOKAHEAD** es superior a 1, entonces JavaCC no realiza dicho control en ninguna opción. Si se da a **FORCE_LA_CHECK** el valor *true* (por defecto vale *false*), el chequeo de prefijo se realizará en todas las opciones incluyendo las que posean **LOOKEAHEAD** local, a pesar de que éste sea correcto.

STATIC: si el valor es *true* (valor por defecto), todos los métodos y variables se crean como estáticas tanto en el analizador léxico como en el sintáctico lo que mejora notablemente su ejecución. Sin embargo, esto hace que sólo pueda realizarse un análisis cada vez e impide que, simultáneamente, distintos objetos de la misma clase puedan procesar diferentes fuentes de entrada. Si se define como estático, un mismo objeto puede procesar diferentes fuentes de manera secuencial, invocando **ReInit()** cada vez que quiera comenzar con una nueva. Si no es estático, pueden construirse (con **new**) diferentes objetos para procesar cada fuente.

DEBUG_PARSER: esta opción se utiliza para obtener información de depuración en tiempo de ejecución del analizador sintáctico generado. Su valor por defecto es *false*, aunque la traza de las acciones y decisiones tomadas puede activarse o desactivarse por programa invocando a **enable_tracing()** y **disable_tracing()** respectivamente. De manera parecida pueden utilizarse las opciones **DEBUG_LOOKAHEAD**, y **DEBUG_TOKEN_MANAGER**.

BUILD_PARSER: indica a JavaCC si debe generar o no la clase del analizador sintáctico. Su valor por defecto es *true*, al igual que el de la opción **BUILD_TOKEN_MANAGER**, lo que hace que se generen ambos analizadores.

IGNORE_CASE: si su valor es *true*, el analizador lexicográfico (*token manager*) generado no efectúa diferencia entre letras mayúsculas y minúsculas a la hora de reconocer los lexemas. Es especialmente útil en el reconocimiento de lenguajes como HTML. Su valor por defecto es *false*.

COMMON_TOKEN_ACTION: si el valor de esta opción es *true*, todas las invocaciones al método **getNextToken()** de la clase del analizador

lexicográfico, causan una llamada al método **CommonTokenAction()** después del reconocimiento de cada lexema y después de ejecutar la acción léxica asociada, si la hay. El usuario debe definir este último método en la sección **TOKEN_MGR_DECLS** del área de *tokens*. El valor por defecto es *false*.

UNICODE_INPUT: si se le da el valor *true* se genera un analizador léxico capaz de gestionar una entrada codificada en Unicode. Por defecto vale *false* lo que asume que la entrada estará en ASCII.

OUTPUT_DIRECTORY: es una opción cuyo valor por defecto es el directorio actual. Controla dónde son generados los archivos de salida.

5.2.2 Área de *tokens*

Este área permite la construcción de un analizador lexicográfico acorde a las necesidades particulares de cada proyecto. En la jerga propia de JavaCC los analizadores léxicos reciben el nombre de *token managers*.

JavaCC diferencia cuatro tipos de *tokens* o terminales, en función de lo que debe hacer con cada lexema asociado a ellos:

- SKIP: ignora el lexema.
- MORE: busca el siguiente el siguiente lexema pero concatenándolo al ya recuperado.
- TOKEN: obtiene un lexema y crea un objeto de tipo **Token** que devuelve al analizador sintáctico (o a quien lo haya invocado). Esta devolución se produce automáticamente sin que el desarrollador deba especificar ninguna sentencia **return** en ninguna acción léxica.
- SPECIAL_TOKEN: igual que SKIP pero almacena los lexemas de tal manera que puedan ser recuperados en caso necesario. Puede ser útil cuando se construyen traductores fuente-fuente de manera que se quieren conservar los comentarios a pesar de que no influyen en el proceso de traducción.

Una vez encontrado un lexema, sea cual sea su tipo, es posible ejecutar una acción léxica.

El formato de esta fase es el siguiente:

```
TOKEN_MGR_DECLS : {
    bloqueJava
}
<estadoLéxico> TOKEN [IGNORE_CASE] : {
        <token1 : patrón1> { acciónLéxica1 } : nuevoEstadoLéxico1
    |   <token2 : patrón2> { acciónLéxica2 } : nuevoEstadoLéxico2
    |       ...
}
```

```
<estadoLéxico> TOKEN [IGNORE_CASE] : {
        <token1 : patrón1> { acciónLéxica1 } : nuevoEstadoLéxico1
    |   <token2 : patrón2> { acciónLéxica2 } : nuevoEstadoLéxico2
    |       ...
}
...
```

donde:

- La palabra TOKEN puede sustituirse por SKIP, MORE o SPECIAL_TOKEN.
- La cláusula [IGNORE_CASE] es opcional pero, si se pone, no deben olvidarse los corchetes.
- El patrón puede ir precedido de un nombre de *token* y separado de éste por dos puntos, pero no es obligatorio (por ejemplo, en los separadores no tendría sentido).
- Si el patrón es una constante entrecomillada sin *token* asociado pueden omitirse los paréntesis angulares que lo engloban y, por tanto, el nombre.
- Los estados léxicos son identificadores cualesquiera, preferentemente en mayúsculas, o bien la palabra DEFAULT que representa el estado léxico inicial. Pueden omitirse. También puede indicarse una lista de estados léxicos separados por comas.
- Si se omite el estado léxico antes de la palabra TOKEN quiere decir que los patrones que agrupa pueden aplicarse en cualquier estado léxico.
- Si se omite el estado léxico tras un patrón quiere decir que, tras encontrar un lexema que encaje con él, se permanecerá en el mismo estado léxico.
- Las acciones léxicas están formadas por código Java y son opcionales.
- Las acciones léxicas tienen acceso a todo lo que se haya declarado dentro de la cláusula opcional TOKEN_MGR_DECLS.
- Las acciones léxicas se ejecutan en el seno de un *token manager* tras recuperar el lexema de que se trate.
- JavaCC sigue las mismas premisas que PCLex a la hora de buscar lexemas, esto es, busca siempre el lexema más largo posible y, en caso de que encaje por dos patrones, opta por el que aparezca en primer lugar.
- Si JavaCC encuentra un lexema que no encaja por ningún patrón produce un error léxico. Esta situación debe ser evitada por el desarrollador mediante el reconocimiento de los lexemas erróneos y la emisión del correspondiente mensaje de error.

5.2.2.1 Caracteres especiales para patrones JavaCC

Los caracteres que tienen un significado especial a la hora de formar expresiones regulares son:

"": sirve para delimitar cualquier cadena de caracteres.

<>: sirve para referenciar un *token* definido previamente. En algunas ocasiones puede resultar interesante crear *tokens* con el único objetivo de utilizarlos dentro de otros, como p.ej.:

```
TOKEN :
{
  < LITERAL_COMA_FLOTANTE:
       (["0"-"9"])+ "," (["0"-"9"])* (<EXPONENTE>)? (["f","F","d","D"])?
     | "." (["0"-"9"])+ (<EXPONENTE>)? (["f","F","d","D"])?
     | (["0"-"9"])+ <EXPONENTE> (["f","F","d","D"])?
     | (["0"-"9"])+ (<EXPONENTE>)? ["f","F","d","D"]
  >
|
  < EXPONENTE: ["e","E"] (["+","-"])? (["0"-"9"])+ >
}
```

El problema de este ejemplo radica en que el analizador léxico puede reconocer la cadena "E123" como del tipo **EXPONENTE**, cuando realmente debiera ser un ID. En situaciones como ésta, el *token* **EXPONENTE** puede declararse como (nótese el uso del carácter "#"):

```
< #EXPONENT: ["e","E"] (["+","-"])? (["0"-"9"])+ >
```

lo que le indica a JavaCC que no se trata de un *token* de pleno derecho, sino que debe utilizarlo como modelo de uso en aquellos otros patrones que lo referencien.

: indica repetición 0 ó mas veces de lo que le precede entre paréntesis. Ej.: ("ab") encaja con ε, "ab", "abab", etc. El patrón "ab"* es incorrecto puesto que el símbolo "*" debe estar precedido por un patrón entre paréntesis.

+: indica repetición 1 ó mas veces de lo que le precede entre paréntesis.

?: indica que lo que le precede entre paréntesis es opcional.

|: indica opcionalidad. Ej.: ("ab" | "cd") encaja con "ab" o con "cd".

[]: sirve para expresar listas de caracteres. En su interior pueden aparecer:

- Caracteres sueltos encerrados entre comillas y separados por comas. Ej.: ["a", "b", "c"] que encajará con los lexemas "a", "b" o "c".
- Rangos de caracteres formados por dos caracteres sueltos entre comillas y separados por un guión. Ej.: ["a"-"c"] que encajará con los lexemas "a", "b" o "c".
- Cualquier combinación de los anteriores. Ej.: ["a"-"z", "A"-"Z", "ñ", "Ñ"] que encajará con cualquier letra española sin signos diacríticos (acentos o diéresis).

~[]: sirve para expresar una lista de caracteres complementaria a la dada según lo visto anteriormente para []. P.ej., el patrón ~[] es una forma de emular al patrón (.|\n) de PCLex.

JavaCC siempre crea automáticamente el *token* <**EOF**> que encaja con el carácter fin de fichero.

Por otro lado, no existe una área de declaración de estados léxicos, como en

PCLex o JFlex, sino que éstos se utilizan directamente. Un sencillo ejemplo para reconocer comentarios no anidados al estilo de Java o C sería:

```
SKIP : {
        "/*" : DentroComentario
}
<DentroComentario> SKIP : {
        "*/" : DEFAULT
}
<DentroComentario> MORE : {
        <~[]>
}
```

5.2.2.2 Elementos accesibles en una acción léxica

El desarrollador puede asociar acciones léxicas a un patrón que serán ejecutadas por el analizador léxico cada vez que encuentre un lexema acorde. En estas acciones se dispone de algunas variables útiles, como son:

image: variable de tipo **StringBuffer** que almacena el lexema actual. Una copia de dicho lexema se almacena también en el campo **token** de la variable **matchedToken**. Si el último *token* aceptado fue de tipo **MORE**, entonces **image** acumulará el lexema actual a lo que ya contuviese. P.ej. si tenemos:

```
<DEFAULT> MORE : {
        "a" {     ❶ ;} : S1
}
<S1> MORE : {
        "b" {     ❷
                  int long = image.length()-1;
                  image.setCharAt(long, image.charAt(long).toUpperCase());
                  ❸
        } : S2
}
<S2> TOKEN : {
        "cd" {    ❹ ; } : DEFAULT
}
```

ante la entrada “abcd”, la variable **image** tendrá los valores “a”, “ab”, “aB” y “aBcd” en los puntos ❶, ❷, ❸ y ❹ del código respectivamente.

lenghtOfMatch: variable de tipo entero que indica la longitud del último lexema recuperado. Siguiendo el ejemplo anterior, esta variable tomaría los valores 1, 1, 1 y 2 en los puntos ❶, ❷, ❸ y ❹ del código respectivamente. La longitud completa de **image** puede calcularse a través del método **length()** de la clase **StringBuffer**.

curLexState: variable de tipo entero que indica el estado léxico actual. Los estados léxicos pueden ser manipulados directamente a través del nombre que el desarrollador les halla dado, ya que JavaCC inicializa las correspondientes constantes Java en uno de los ficheros que genera (**EjemploConstants.java**

según el ejemplo del apartado 5.1.2).

matchedToken: variable de tipo **Token** que almacena el *token* actual. Siguiendo el ejemplo anterior, **matchedToken.token** tomaría los valores "a", "ab", "ab" y "abcd" en los puntos ❶, ❷, ❸ y ❹ del código respectivamente.

switchTo(int estadoLéxico): función que permite cambiar por programa a un nuevo estado léxico. Si se invoca esta función desde una acción léxica debe ser lo último que ésta ejecute. Evidentemente, para que tenga efecto no debe usarse la cláusula **: nuevoEstadoLéxico1** tras la acción léxica. Esta función puede resultar muy útil en el reconocimiento, p. ej., de comentarios anidados:

```
TOKEN_MGR_DECLS : {
        int numComentarios = 0;
}
SKIP: {
        "/*" { numComentarios= 1;} : DentroComentario
}
<DentroComentario> SKIP: {
                "/*" { numComentarios++;}
        |       "*/" { numComentarios--; if (numComentarios == 0)
                switchTo(DEFAULT); }
        |       <(~[\*\/\t\n\r])+>
        |       <[\*\/\t\n\t]>
}
```

Las acciones léxicas también pueden utilizar todo lo que se haya declarado en la cláusula opcional **TOKEN_MGR_DECLS**. el siguiente ejemplo visualiza la longitud de una cadena entrecomillada haciendo uso de varios patrones y una variable común **longCadena** (nótese la importancia del orden de los dos últimos patrones ya que, de otra forma nunca se reconocerían las comillas de cierre):

```
TOKEN_MGR_DECLS : {
        int longCadena;
}
MORE : {
        "\"" { longCadena= 0;} : DentroCadena
}
<DentroCadena> TOKEN : {
        <STRLIT: "\""> {System.out.println("Size = " + longCadena);} :
DEFAULT
}
<DentroCadena> MORE : {
        <~["\n","\r"]> {longCadena++;}
}
```

Si la opción **COMMON_TOKEN_ACTION** es *true* debe especificarse obligatoriamente la clásula **TOKEN_MGR_DECLS** que, como mínimo, deberá contener la función **CommonTokenAction()**, cuya cabecera es:

```
void CommonTokenAction(Token t)
```

y que será invocada cada vez que se recupere cualquier lexema sea cual sea su tipo, tal y como se vio en el apartado 5.2.1.

5.2.2.3 La clase Token y el *token manager*

Como ya se ha comentado, cada vez que el analizador léxico encuentra un lexema correspondiente a un *token* normal, retorna un objeto de la clase **Token**. Las variables y funciones principales de esta clase son:

- **image**: variable de tipo **String** que almacena el lexema reconocido.
- **kind**: para cada patrón al que el desarrollador le asocia un nombre, JavaCC incluye una constante de tipo entera en la clase de constantes que genera. La variable **kind** almacena uno de dichos enteros, concretamente el del nombre del patrón por el que encajó su lexema.
- **beginLine**, **beginColumn**, **endLine** y **endColumn**: variables de tipo entero que indican, respectivamente la línea y columna de comienzo y de final del lexema.
- **specialToken**: variable de tipo **Token** que referencia al token anterior si y sólo si éste era de tipo especial; en caso contrario almacenará el valor **null**.

Para finalizar, indicar que JavaCC proporciona en la clase del analizador léxico (**EjemploTokenManager** según el ejemplo del punto 5.1.2) dos métodos que pueden ser interesantes, caso de querer construir tan sólo dicho analizador, a saber:

- Un constructor que toma como parámetro un objeto de tipo **CharStream** del que tomará la entrada a analizar.
- La función **getNextToken()** que devuelve el siguiente objeto de tipo **Token** que encuentra en la entrada. El desarrollador debe controlar el tipo de *token* <**EOF**> para no consumir más allá del fin de la entrada.

El siguiente ejemplo ilustra un programa JavaCC del que sólo se pretende emplear la parte de analizador léxico. A pesar de que la opción **BUILD_PARSER** está a *false*, JavaCC obliga a indicar las cláusulas **PARSER_BEGIN** y **PARSER_END**, y una clase del mismo nombre en su interior, aunque esté vacía.

```
options{
    /* No generar la clase Ejemplo */
    BUILD_PARSER=false;
}
PARSER_BEGIN(Ejemplo)
    public class Ejemplo{}
PARSER_END(Ejemplo)
/* Declaración de la función main() en el token manager */
TOKEN_MGR_DECLS : {
    public static void main(String args[]) throws ParseException {
        EjemploTokenManager miLexer =
            new EjemploTokenManager(new SimpleCharStream(System.in));
```

```
            Token t;
            while((t=miLexer.getNextToken()).kind != EjemploTokenManager.EOF)
                System.out.println("Vd. ha escrito"+t.image);
        }
}
SKIP : {
            " "
        |   "\t"
        |   "\n"
        |   "\r"
}
TOKEN [IGNORE_CASE] : {
        <ID: (["a"-"z"])+>
}
```

De esta manera, la función **main()** se incluye directamente en el *token manager* mediante la cláusula **TOKEN_MGR_DECLS**, y en ella construimos un canal de entrada del tipo esperado por el analizador léxico, a quien se lo pasamos en su constructor. Los canales de entrada de los que puede leer el *token manager* son del tipo **CharStream**; JavaCC proporciona la clase **SimpleCharStream** que hereda de **CharStream** para facilitar la creación de canales aceptables para ser analizados lexicográficamente. **SimpleCharStream** es un envoltorio o estrato para las clases **InputStream** y **Reader** poseyendo constructores que toman como parámetros objetos de cualesquiera de estos dos tipos.

Una vez hecho esto, los *tokens* se van recuperando de uno en uno visualizando por pantalla el correspondiente mensaje. El mismo efecto se podría haber obtenido escribiendo:

```
options{ BUILD_PARSER=false; }
PARSER_BEGIN(Ejemplo) public class Ejemplo{} PARSER_END(Ejemplo)
TOKEN_MGR_DECLS : {
   public static void main(String args[]) throws ParseException {
      new EjemploTokenManager(new SimpleCharStream(System.in)).getNextToken();
   }
}
SKIP : {
      " "
   |  "\t"
   |  "\n"
   |  "\r"
}
SKIP [IGNORE_CASE] : {
   <ID: (["a"-"z"])+> { System.out.println("Vd. ha escrito"+image); }
}
```

mucho más compacto y donde se han resaltado las modificaciones con respecto al equivalente anterior.

5.2.3 Área de funciones BNF

Como ya se ha comentado, JavaCC genera un analizador sintáctico descendente implementado a base de funciones recursivas, de manera que cada no terminal de nuestra gramática será convertido una función diferente, cuya implementación será generada por JavaCC.

Este área tiene la siguiente estructura:

$tipoRetorno_1$ $funcionJava_1(param_1)$:
    { $codigoJava_1$ }
    { $exprBNF_1$ }
$tipoRetorno_2$ $funcionJava_2(param_2)$:
    { $codigoJava_2$ }
    { $exprBNF_2$ }
    ...

Dado que cada no terminal se convertirá en una función en Java, el desarrollador no sólo debe indicar su nombre, sino la cabecera completa de dicha función. Este hecho es de fundamental importancia puesto que:

> «JavaCC permite el intercambio de atributos entre reglas BNF mediante el paso de parámetros y la obtención de resultados en el momento de hacer uso de un no terminal (o lo que es lo mismo, invocar a la función que lo implementa) lo que equivale, respectivamente, a enviar atributos hacia abajo y hacia arriba en el árbol sintáctico».

Por otro lado, las expresiones BNF **$exprBNF_i$** pueden hacer uso de los siguientes componentes:

noTerminal(): representa la utilización del no terminal **noTerminal**. Nótese la utilización de los paréntesis de invocación de función.

<>: permiten hacer referencia a un terminal declarado en el área de *tokens*. Ej.: <NUMERO>

“”: permiten expresar *tokens* anónimos, entrecomillando un literal de tipo cadena de caracteres.

*: indica repetición 0 ó mas veces de lo que le precede entre paréntesis.

+: indica repetición 1 ó mas veces de lo que le precede entre paréntesis.

?: indica que lo que le precede entre paréntesis es opcional.

[]: tiene igual significado que el ?, de manera que (patron)? es equivalente a [patron].

|: indica opcionalidad. Ej.: (“texto1” | “texto2”) encaja con “texto1” y con “texto2”.

De esta manera, una forma para reconocer expresiones vendría dada por la gramática:

```
void expr():{}{
```

```
        term() ( (“+”|”-“) term() )*
    }
    void term():{}{
        fact() ( (“*”|”/“) fact() )*
    }
    void fact():{}{
                <NUMERO>
        |       <ID>
        |       “(“ expr() “)”
    }
```

El apartado **codigoJava**$_i$ permite ubicar declaraciones y sentencias Java al comienzo de la implementación de una función no terminal. Entre otras cosas, esto nos puede permitir obtener una traza de las acciones que toma nuestro analizador sintáctico para reconocer una sentencia. Por ejemplo, el programa:

```
        void expr():{ System.out.println(“Reconociendo una expresión”); }{
                term() ( (“+”|”-“) term() )*
        }
        void term():{ System.out.println(“Reconociendo un término”); }{
                fact() ( (“*”|”/“) fact() )*
        }
        void fact():{ System.out.println(“Reconociendo un factor”); }{
                        <NUMERO>
                |       <ID>
                |       “(“ expr() “)”
        }
```

ante la entrada “23+5*(8-5)” produciría la salida:

```
Reconociendo una expresión
Reconociendo un término
Reconociendo un factor
Reconociendo un término
Reconociendo un factor
Reconociendo un factor
Reconociendo una expresión
Reconociendo un término
Reconociendo un factor
Reconociendo un término
Reconociendo un factor
```

lo que coincide con el recorrido en preorden del árbol sintáctico que reconoce dicha

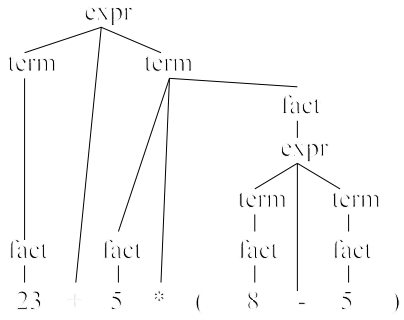

Figure 2Árbol sintáctico que reconoce una expresión en base a reglas BNF. Para mayor claridad no se han dibujado los *tokens* NUMERO

sentencia, y que se muestra en la figura 5.2.

Nótese que cuando se utiliza notación BNF, el número de símbolos bajo una misma regla puede variar si en ésta se emplea la repetición. P.ej., siguiendo con la gramática anterior, la regla de la **expr** puede dar lugar a tantos hijos como se quiera en el árbol sintáctico; sirva como demostración el árbol asociado al reconocimiento de "23+5+7+8+5", que se ilustra en la figura 5.3, y cuya corroboración viene dada por la salida del analizador sintáctico:

```
Reconociendo una expresión
Reconociendo un término
Reconociendo un factor
Reconociendo un término
Reconociendo un factor
Reconociendo un término
Reconociendo un factor
Reconociendo un término
Reconociendo un factor
Reconociendo un término
Reconociendo un factor
```

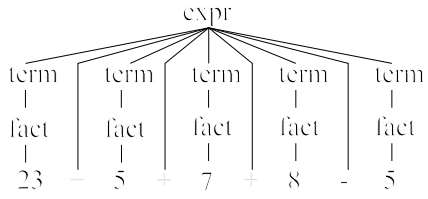

Figure 3 El árbol sintáctico asociado a una expresión BNF con expansiones del tipo ***** ó **+** puede tener un número indeterminado de hijos

Como ya se ha comentado, JavaCC genera por defecto analizadores de tipo LL(1), lo que quiere decir que son capaces de escoger entre diferentes opciones de flujo consultando tan sólo el siguiente *token* de la entrada. Sin embargo, ya sabemos que no todas las gramáticas son LL(1), por lo que puede haber situaciones en las que sea necesario consultar más de un *token* de la entrada para poder tomar una decisión. Por ejemplo, la siguiente gramática:

```
void expr():{}{
                <ID>
        |       <ID> "." expr()
        |       <ID> "()"
}
```

no es LL(1) porque, ante una entrada como “cont.aux()” el analizador sintáctico se encuentra como primer *token* con un **<ID>**, pero como las tres posibles opciones de **expr()** comienzan por **<ID>**, pues se carece de suficiente información para saber por cuál de ellas optar. En este caso, una prebúsqueda de dos *tokens* nos permitiría decidir correctamente.

Para ello, la opción **LOOKAHEAD** vista en el apartado 5.2.1 permite indicarle a JavaCC el tipo de gramática que se le está proporcionando, esto es, el número de *tokens* que debe prebuscar antes de tomar una decisión.

No obstante lo anterior, en la mayoría de los casos, las gramáticas de los lenguajes de programación contienen una mayoría de expresiones LL(1), y tan sólo unas pocas reglas para las cuales puede ser necesario una prebúsqueda de 2, o 3 a lo sumo. En estos casos, se obtendrá un analizador sintáctico más eficiente si se deja la prebúsqueda por defecto (a 1), y se le indica a JavaCC en qué casos concretos debe emplear una prebúsqueda superior. Esto se consigue con la prebúsqueda o *lookahead* local, que tiene la forma:

LOOKAHEAD(**n**) $patron_1$

y que es considerado en sí mismo un patrón también. El significado de esta cláusula es muy sencillo: le indica al analizador sintáctico que, antes de decidirse a entrar por el $\mathbf{patron_1}$ se asegure de que éste es el correcto tomando **n** *tokens*. De esta manera, la gramática anterior se puede dejar como LL(1) e indicar la prebúsqueda local de la forma:

```
void expr():{}{
                LOOKAHEAD(2) <ID>
        |       LOOKAHEAD(2) <ID> "." expr()
        |       <ID> "()"
}
```

Además, el propio JavaCC es capaz de decidir si la prebúsqueda especificada por el desarrollador es adecuada o no, informando de ello en caso negativo tanto por exceso como por defecto.

5.3 Gestión de atributos.

JavaCC carece de una gestión de atributos específica, pero permite el paso de parámetros y la devolución de valores cada vez que se invoca a un no terminal, tal y como se introdujo en el punto 5.2.3. De esta manera, cada vez que se invoca a un no terminal, podemos asignar el valor que devuelve a una variable global declarada en el bloque de código Java asociado a dicha regla; este valor hace las veces de atributo sintetizado. Además, cuando se declara un no terminal, puede poseer parámetros formales, de tal manera que deba ser invocada con parámetros reales que harán las veces de atributos heredados.

Desgraciadamente, JavaCC tampoco permite asociar atributos a los terminales, por lo que deberán ser las acciones semánticas quienes trabajen directamente con los lexemas. Aunque esto puede ser un engorro en la mayoría de los casos, tiene la ventaja de que cada acción semántica puede extraer del lexema la información que le convenga, en lugar de utilizar siempre el mismo atributo en todos los consecuentes en los que parezca un mismo terminal. El siguiente ejemplo ilustra las reglas BNF y las acciones semánticas asociadas para crear una calculadora:

```
int expr():{
      int   acum1=0,
            acum2=0;
}{
      acum1=term()      (     ❶("+" acum2=term() {acum1+=acum2;} )
                        |     ❷("-" acum2=term() {acum1-=acum2;} )
                        )*
      { return acum1; }
}
int term():{
      int   acum1=0,
            acum2=0;
}{
      acum1=fact()      (     ❸("*" acum2=fact() {acum1*=acum2;} )
                        |     ❹("/" acum2=fact() {acum1/=acum2;} )
                        )*
      { return acum1; }
}
int fact():{
      int acum=0;
}{
            <NUMERO>            { return Integer.parseInt(token.image); }
      |     "(" acum=expr() ")"     { return acum; }
}
```

Como puede observarse en este ejemplo, cuando un no terminal devuelve algún valor, debe incluirse una acción semántica al final de cada una de las opciones que lo definen, con una sentencia **return** coherente. Además, es posible colocar acciones semánticas en el interior de las subexpresiones BNF (como en el caso de **expr** y **term**, en los puntos ❶, ❷, ❸ y ❹). Una acción semántica se ejecutará cada vez que el flujo de reconocimiento pase por ellas. Por ejemplo, ante la entrada 23+5+11 se pasará 2 veces por el punto ❶, ya que es necesario pasar 2 veces por la subexpresión ("+" term())* para reconocer dicha entrada.

Además, nótese que una acción semántica colocada justo después de un terminal puede acceder al objeto *token* asociado al mismo, a través de la variable **token**. Esto es así a pesar de la prebúsqueda que se haya indicado, ya que es el analizador sintáctico el que carga dicha variable justo después de consumir cada terminal, e independientemente de los que haya tenido que prebuscar para tomar una decisión sobre qué opción tomar.

Cuando se incluyen acciones semánticas en el interior de una expresión BNF, ésta se enrarece sobremanera, siendo muy difícil reconocer la gramática en un mar de código. Para evitar este problema se recomienda colocar en un comentario delante de cada no terminal la expresión BNF pura que lo define.

Por otro lado, cuando conviene que un terminal **t** tenga siempre el mismo atributo en todos los consecuentes en los que aparece, suele resultar más legible crear

una regla **T::=t**, asociarle a **T** el atributo calculado a partir de **t** y, en el resto de reglas, usar **T** en lugar de **t**.

Siguiendo los dos consejos anteriores, el ejemplo de la calculadora quedaría:

```
/*
expr ::= term ( ( "+" | "-" ) term )*
*/
int expr():{
      int   acum1=0,
            acum2=0;
}{
      acum1=term()      (     ❶("+" acum2=term() {acum1+=acum2;} )
                        |     ❷("-" acum2=term() {acum1-=acum2;} )
                        )*
      { return acum1; }
}
/*
term ::= fact ( ( "*" | "/" ) fact )*
*/
int term():{
      int   acum1=0,
            acum2=0;
}{
      acum1=fact()      (     ❸("*" acum2=fact() {acum1*=acum2;} )
                        |     ❹("/" acum2=fact() {acum1/=acum2;} )
                        )*
      { return acum1; }
}
/*
fact ::= NUMERO | "(" expr ")"
*/
int fact():{
      int acum=0;
}{
            acum=numero()         { return acum; }
      |     "(" acum=expr() ")"   { return acum; }
}
int numero(): {}{
      <NUMERO> { return Integer.parseInt(token.image); }
}
```

5.4 Gestión de errores.

La gestión de errores puede conseguirse en JavaCC en dos vertientes:

- Personalizando los mensajes de error.
- Recuperándose de los errores.

5.4.1 Versiones antiguas de JavaCC

Hasta la versión 0.6 de JavaCC para personalizar los mensajes de error era necesario establecer la opción **ERROR_REPORTING** a *true*. Con ello JavaCC proveerá a la clase del analizador sintáctico con las variables:

- **error_line**, **error_column**: variables de tipo entero que representan la línea y la columna en la que se ha detectado el error.
- **error_string**: variable de tipo **String** que almacena el lexema que ha producido el error. Si la prebúsqueda (*lookahead*) se ha establecido a un valor superior a 1, **error_string** puede contener una secuencia de lexemas.
- **expected_tokens**: variable de tipo **String[]** que contiene los *tokens* que habrían sido válidos. De nuevo, si la prebúsqueda se ha establecido a un valor superior a 1, cada ítem de **expected_tokens** puede ser una secuencia de *tokens* en lugar de un *token* suelto.

Cada vez que el analizador sintáctico se encuentre un error sintáctico invocará a la función **token_error(),** que deberá ser re-escrita por el desarrollador por lo que éste deberá crearse una clase nueva que herede de la clase del analizador sintáctico.

De manera parecida, en el analizador léxico pueden hacerse uso de los elementos:

- **error_line**, **error_column**: línea y columna del error léxico.
- **error_after**: cadena de caracteres que contiene el trozo de lexema que se intentaba reconocer.
- **curChar**: carácter que ha producido el error léxico.
- **LexicalError()**: función invocada por el *token manager* cuando se encuentra un error léxico.

5.4.2 Versiones modernas de JavaCC

Actualmente, JavaCC gestiona los errores léxicos y sintácticos lanzando las excepciones **TokenMgrError** y **ParseException**, cuyas clases genera JavaCC automáticamente si no existen ya en el directorio de trabajo. La filosofía es que el desarrollador debe impedir que se produzcan errores léxicos, controlando adecuadamente cualquier lexema de entrada inclusive los no válidos, en cuyo caso deberá emitir el correspondiente mensaje de error.

De esta manera, el desarrollador puede modificar la clase **ParseException** para adaptarla a sus necesidades. Por defecto, esta clase posee las variables:

- **currentToken**: de tipo **Token**, almacena el objeto *token* que ha producido el error.
- **expectedTokenSequences**: de tipo **int[][]** tiene el mismo objetivo que la variable **expected_tokens** vista en el apartado anterior.

- **tokenImage**: de tipo **String[]** lmacena los últimos lexemas leídos. Si la prebúsqueda está a 1, entonces se cumple:
  tokenImage[0].equals(currentToken.image);

Por otro lado, la recuperación de errores no requiere modificación alguna de la clase **ParseException**, sino que para ello se utiliza una cláusula similar a la **try-catch** de Java para capturar la excepción en el punto que más convenga. Dicha cláusula se considera asimismo un patrón, por lo que puede aparecer en cualquier lugar donde se espere uno. La cláusula es de la forma:

```
try {
        patron
        catch(CualquierExcepcion x){
                codigoDeRecuperacionJava
        }
}
```

Nótese que la cláusula **try-catch** de JavaCC puede capturar cualquier excepción, y no sólo las de tipo **ParseException**, por lo que el desarrollador puede crear sus propias excepciones y elevarlas dónde y cuando se produzcan. Por ejemplo, puede resultar útil el crearse la excepción **TablaDeSimbolosException**.

El código de recuperación suele trabajar en *panic mode*, consumiendo toda la entrada hasta llegar a un *token* de seguridad. El siguiente ejemplo ilustra cómo recuperar errores en sentencias acabadas en punto y coma.

```
void sentenciaFinalizada() : {} {
        try {       ( sentencia() <PUNTOYCOMA> )
        catch (ParseException e) {
                System.out.println(e.toString());
                Token t;
                do {
                        t = getNextToken();
                } while (t.kind != PUNTOYCOMA);
        }
}
```

5.5 Ejemplo final

Ensamblando las distintas partes que se han ido viendo a lo largo de este capítulo, y observando los consejos dados en cuanto a comentarios, el ejemplo completo de la calculadora con control de errores quedaría:

```
PARSER_BEGIN(Calculadora)
    public class Calculadora{
      public static void main(String args[]) throws ParseException {
    new Calculadora(System.in).gramatica();
    }
    }
PARSER_END(Calculadora)
SKIP : {
```

```
            " "
      |     "\t"
      |     "\n"
      |     "\r"
}
TOKEN [IGNORE_CASE] :
{
            <NUMERO: (["0"-"9"])+>
      |     <PUNTOYCOMA: ";">
}
/*
gramatica ::= ( exprFinalizada )+
*/
void gramatica():{}{
      ( exprFinalizada() )+
}
/*
exprFinalizada ::= expr ";"
*/
void exprFinalizada():{
      int resultado;
}{
      try{
            resultado=expr() <PUNTOYCOMA> {System.out.println("="+resultado); }
      }catch(ParseException x){
            System.out.println(x.toString());
            Token t;
            do {
                  t = getNextToken();
            } while (t.kind != PUNTOYCOMA);
      }
}
/*
expr ::= term ( ( "+" | "-" ) term )*
*/
int expr():{
      int   acum1=0,
            acum2=0;
}{
      acum1=term()      (     ("+" acum2=term() {acum1+=acum2;} )
                        |     ("-" acum2=term() {acum1-=acum2;} )
                        )*
      { return acum1; }
}
/*
term ::= fact ( ( "*" | "/" ) fact )*
*/
int term():{
      int   acum1=0,
```

```
        acum2=0;
}{
    acum1=fact()    (   ("*" acum2=fact() {acum1*=acum2;} )
                    |   ("/" acum2=fact() {acum1/=acum2;} )
                    )*
    { return acum1; }
}
/*
fact ::= NUMERO | "(" expr ")"
*/
int fact():{
    int acum=0,
        uMenos=1;
}{
    ("-" { uMenos = -1; } )?
    (   <NUMERO>            { return Integer.parseInt(token.image)*uMenos; }
    |   "(" acum=expr() ")"     { return acum*uMenos; }
    )
}
SKIP : {
    <ILEGAL: (~[])>   { System.out.println("Carácter: "+image+" no esperado.");}
}
```

Nótese cómo se ha tenido que definir el terminal **PUNTOYCOMA** para poder hacer referencia a él en la gestión del error.

Por otro lado, el orden de las expresiones regulares es fundamental, de tal manera que la regla:

```
<ILEGAL: (~[])>
```

recoge todos los lexemas que no hayan entrado por ninguna de las reglas anteriores, incluidas las *ad hoc* especificadas en la propia gramática BNF.

5.6 La utilidad JJTree

JJTree es el preprocesador para JavaCC. Inserta acciones que construyen árboles parser en el fuente JavaCC. La salida generada por JJTree se ejecuta a través de JavaCC para crear el parser.

5.6.1 Características

Por defecto, JJTree genera código para construir el árbol de nodos del parse para cada no terminal de nuestro lenguaje. Este funcionamiento puede modificarse de manera que ciertos no terminales no generen nodos, o generar un nodo a partir de una expansión de una regla de producción.

JJTree define una interfaz de Java tipo **Node,** el cual todos los nodos del árbol parser deben implementar. La interfaz proporciona métodos para operaciones como inicialización del tipo del padre de un nodo generado, así como adición y recuperación de hijos.

JJTree puede operar de dos formas distintas: simple o multi (para buscar términos mejores). En modo simple, cada nodo del árbol parser es del tipo **SimpleNode**; en modo multi, el tipo de los nodos del árbol parse se deriva del nombre del nodo. Si no se proporciona implementaciones para la clase nodo, JJTree generará implementaciones basadas en la clase **SimpleNode**, pudiendo modificar las implementaciones de manera que puedan sernos de utilidad.

5.6.2 Ejemplo Práctico

Al final del presente documento hay una sección (Pasos de Ejecución) en la que se indica cuáles son los pasos que hay que dar para la compilación de ficheros **.jjt** y **.jj.** Un avance se muestra a continuación.

Para la ejecución de JJTree es necesario alimentar la entrada de la siguiente manera:

```
jjtree ejemplo1.jjt
```

Todos los archivos que se generan de forma automática se pueden consultar en la sección Ficheros Fuente.

Supongamos que creamos un archivo llamado **ejemplo1.jjt** con las siguientes especificaciones:

```
PARSER_BEGIN(ejemplo1)
class ejemplo1 {
	public static void main(String args[]) {
		System.out.println("Entrada teclado");
		ejemplo1 ins = new ejemplo1(System.in);
		try {
			SimpleNode nodo = ins.Inicio();
			nodo.dump("");
		} catch (Exception e) {
			System.out.println(e.getMessage());
			e.printStackTrace();
		}
	}
}
PARSER_END(ejemplo1)
SKIP :{
		" "
	|	 "\t"
	|	 "\n"
	|	 "\r"
	|	 <"//" (~["\n","\r"])* ("\n"|"\r"|"\r\n")>
	|	 <"/*" (~["*"])* "*" (~["/"] (~["*"])* "*")* "/">
}

TOKEN : /* LITERALES */
{
```

```
<ENTERO:
        <DECIMAL>
        |       <HEX>
        |       <OCTAL>
>
|
  < #DECIMAL: ["1"-"9"] (["0"-"9"])* >
|
  < #HEX: "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+ >
|
  < #OCTAL: "0" (["0"-"7"])* >
}

TOKEN : /* IDENTIFICADORES */
{
  < IDENTIFICADOR: <LETRA> (<LETRA>|<DIGITO>)* >
|
  < #LETRA: ["_","a"-"z","A"-"Z"] >
|
  < #DIGITO: ["0"-"9"] >
}

SimpleNode Inicio() : {}
{
        Expresion() ";"
        { return jjtThis; }
}
void Expresion() : {}
{
        ExpSuma()
}
void ExpSuma() : {}
{
        ExpMult() ( ( "+" | "-" ) ExpMult() )*
}
void ExpMult() : {}
{
        ExpUnaria() ( ( "*" | "/" | "%" ) ExpUnaria() )*
}
void ExpUnaria() : {}
{
         "(" Expresion() ")" | Identificador() | <ENTERO>
}
void Identificador() : {}
{
        <IDENTIFICADOR>
}
```

La sentencia **e.printStackTrace()**; me indica que obtengo el contenido de la

pila cuando ha ocurrido un error. Una traza de ejemplo se muestra a continuación; en este caso, el error ocurrido es el acceso a una zona de memoria que está a null:

```
...
java.lang.NullPointerException
        at ejemplo4.Identificador(ejemplo4.java:228)
        at ejemplo4.ExpUnaria(ejemplo4.java:200)
        at ejemplo4.ExpMult(ejemplo4.java:139)
        at ejemplo4.ExpSuma(ejemplo4.java:84)
        at ejemplo4.Expresion(ejemplo4.java:75)
        at ejemplo4.ExpUnaria(ejemplo4.java:196)
        at ejemplo4.ExpMult(ejemplo4.java:139)
        at ejemplo4.ExpSuma(ejemplo4.java:84)
        at ejemplo4.Expresion(ejemplo4.java:75)
        at ejemplo4.Inicio(ejemplo4.java:45)
        at ejemplo4.Main(ejemplo4.java:30)
```

Por otro lado, **dump()** es un método de la clase **SimpleNode** que se genera automáticamente, y que hace posible un recorrido automático por todos los nodos hijo del padre que se le pasa como parámetro. La cadena de caracteres pasada al método indica el prefijo que se le añade a los nodos cuando se muestran por pantalla. Todo esto se explica a lo largo del documento.

Si ejecutamos **jjtree ejemplo1.jjt** obtenemos cuatro ficheros:

- **Node.java**,
- **SimpleNode.java**,
- **ejemplo1TreeConstants.java** y
- **JJTejemplo1State.java**

El contenido de **ejemplo1TreeConstants.java** es el siguiente:

```
public interface ejemplo1TreeConstants {
        public int JJTINICIO = 0;
        public int JJTEXPRESION = 1;
        public int JJTEXPSUMA = 2;
        public int JJTEXPMULT = 3;
        public int JJTEXPUNARIA = 4;
        public int JJTIDENTIFICADOR = 5;
        public String[] jjtNodeName = {
                "Inicio",
                "Expresion",
                "ExpSuma",
                "ExpMult",
                "ExpUnaria",
                "Identificador",
        };
}
```

Una vez hecho esto, si queremos que nuestra gramática reconozca una sentencia introducida por la entrada asignada en la cabecera del fuente (en este caso el

teclado), hemos de ejecutar el siguiente comando

javacc ejemplo1.jj

El archivo **ejemplo1.jj** ha sido generado de forma automática al reconocer JJTree nuestra gramática como correcta.

Los archivos generados automáticamente por **javacc** para cualquier fuente **.jj** están especificados en la sección **Ficheros Fuente**.

Compilamos con **javac** el fichero .java que hemos obtenido. Dicho ejecutable lo encontramos en la versión 1.5 de JDK de Java. Tras este paso, obtenemos el ejecutable que lanzará la lectura de la cadena a reconocer:

javac ejemplo1.java
java ejemplo1

Si le proporcionamos la entrada siguiente

(id + id) * (id + id)

el árbol generado es el siguiente:

```
Inicio
Expresion
  ExpSuma
    ExpMult
      ExpUnaria
        Expresion
          ExpSuma
            ExpMult
              ExpUnaria
                Identificador
            ExpMult
              ExpUnaria
            Identificador
              ExpUnaria
                Expresion
                  ExpSuma
                    ExpMult
                      ExpUnaria
                        Identificador
                    ExpMult
                      ExpUnaria
                        Identificador
```

Vemos que se le asigna un valor de tipo entero a cada no-terminal (función java en el archivo **.jjt**) de forma única, y que por defecto, se le asigna un nombre a cada nodo (no-terminal). Dicho nombre, si no es asignado por el usuario, es el mismo que el nombre del método que implementa al no-terminal. Para poder asignarle un nombre distinto al que se le pone por defecto, es necesario utilizar el carácter **#** seguido del nombre que le queremos asignar.

5.6.2.1 Renombrado de nodos mediante

es un constructor que se utiliza para que el usuario pueda llamar a cada nodo de una forma distinta a como JavaCC le llama por defecto. Ha de operar en modo multi para que cada nodo pueda derivarse del nombre que pongamos inmediatamente después del constructor #.Sirve también para simplificar la sintaxis y eliminar de la pila los nodos que no aportan información al árbol generado por nuestra gramática.

En la zona de opciones, el **modo multi** debe estar **activo** de la siguiente manera:

```
options {
        MULTI=true;
}
```

Si utilizamos esto último para el mismo ejemplo (**ejemplo1bis.jjt**), la sintaxis sería:

```
void Expresion() #void : {}
{
        ExpSuma()
}
void ExpSuma() #void : {}
{
        (ExpMult() ( ( "+" | "-" ) ExpMult() )*)#Suma(>1)
}
void ExpMult() #void: {}
{
        (ExpUnaria()(("*"|"/"|"%")ExpUnaria())*)#Mult(>1)
}
void ExpUnaria() #void : {}
{
        "(" Expresion() ")" | Identificador() | <LITERAL_ENTERO>
}
```

Y el archivo **ejemplo1bisTreeConstans** quedaría:

```
public interface ejemplo1bisTreeConstants {
        public int JJTINICIO = 0;
        public int JJTVOID = 1;
        public int JJTSUMA = 2;
        public int JJTMULT = 3;
        public int JJTIDENTIFICADOR = 4;
        public String[] jjtNodeName = {
                "Inicio",
                "void",
                "Suma",
                "Mult",
                "Identificador",
        };
}
```

Alimentando de nuevo la entrada con la misma cadena, obtenemos la siguiente salida: (traza de la pila)

```
Inicio
Mult
        Suma
                Identificador
                Identificador
        Suma
                Identificador
                Identificador
```

Observamos que aquellos no-terminales a los que hemos renombrado como **#void,** no aparecen en la salida correspondiente a la pila. Esto es muy útil cuando queremos utilizar sólo los nodos que realmente aportan información al árbol, como ya se ha comentado anteriormente.

5.6.2.2 Construcción del Árbol

Aunque JavaCC es un generador de gramáticas descendente, JJTree construye el árbol de la secuencia de reglas gramaticales desde abajo, es decir, de forma ascendente. Para ello, hace uso de una pila donde se almacenan los nodos una vez generados. Cuando se intenta reducir un conjunto de nodos mediante la asignación de un nodo padre para todos ellos, dichos nodos hijos se sacan de la pila y se le asocia al padre, almacenando posteriormente el conjunto padre-hijos en la pila. Siempre que la pila esté accesible (pila abierta) se podrá efectuar operaciones en el ámbito de las acciones gramaticales tales como 'push','pop' y cualquier otra que se considere oportuna a nuestras necesidades.

Estos métodos se implementan de forma automática, ya que JJTree es el que se encarga de generar el fichero de estado arriba mencionado (para nuestro ejemplo, **JJTejemplo1State.java** y **JJTejemplo1bisState.java**; ver Estados en JJTree para más información). A continuación se muestran algunos métodos que han sido generados automáticamente:

```
JJTejemplo1State() {
        nodes = new java.util.Stack();
        marks = new java.util.Stack();
        sp = 0;
        mk = 0;
}
boolean nodeCreated() {
        return node_created;
}
Node rootNode() {
        return (Node)nodes.elementAt(0);
}
void pushNode(Node n) {
        nodes.push(n);
        ++sp;
}
Node popNode() {
```

```
        if (--sp < mk) {
                mk = ((Integer)marks.pop()).intValue();
        }
        return (Node)nodes.pop();
}
```

5.6.2.3 Tipos de Nodos

JJTree proporciona decoraciones para dos tipos básicos de nodos, además de algunas simplificaciones sintácticas para facilitar su uso.

1. Un nodo se construye con un número definido de hijos. Recordemos que a un nodo padre se le asocian uno o más nodos hijos que son sacados de la pila e introducidos de nuevo en ella, pero formando parte de la estructura del padre. Un nodo definido se puede crear de la siguiente manera:

   #ADefiniteNode (INTEGER EXPRESSION)

   INTEGER EXPRESSION puede ser cualquier tipo de expresión entera. Lo más común es que dicha expresión sea una constante (de tipo entero). **ADefiniteNode** es el nombre con el cual queremos designar al nodo (no-terminal, función java) que le precede.

2. Un nodo condicional se construye con todos los nodos hijos asociados al ámbito del padre siempre que la condición del padre se evalúe a true. Si no es así, dicho nodo no se crea, devolviendo a la pila los nodos hijo que previamente habían sido asociados al nodo padre evaluado. Esto se representa de la siguiente manera:

   #ConditionalNode (BOOLEAN EXPRESSION)

   BOOLEAN EXPRESSION puede ser cualquier expresión de tipo boolean. Igual que en el caso anterior, el nombre que se le da al nodo viene dado por **ConditionalNode**. Existen dos tipos comunes de abreviaturas para dichos nodos:

   a. **#IndefiniteNode** es equivalente a **#IndefiniteNode (true)**

   b. **#GTNode (>1)** es equivalente a **#GTNode (JJTree.arity () > 1)**

   #IndefiniteNode puede llevarnos a situaciones ambiguas en JJTree cuando dicha expresión le sigue otra con paréntesis. En estos casos, se debe sustituir la expresión reducida por su versión sin reducir:

   (…..) #N (a ()) es una expresión ambigua. En su lugar, habría que utilizar (…..) #N (true) (a ())

   Nótese que las expresiones pasadas como parámetros a los nodos no deben tener efectos laterales. JJTree no especifica cuántas veces será evaluada la expresión.

Por defecto, JJTree trata cada no terminal como un nodo indefinido y deriva su nombre a partir de su producción. El usuario puede cambiar este nombre derivado mediante la siguiente expresión, tal y como quedó expuesto más arriba, en la zona del ejemplo:

```
void ExpSuma() #expS : {}
{
        ExpMult() ( ( "+" | "-" ) ExpMult() )*
}
```

Cuando el parser reconoce al no-terminal **ExpSuma()**, comienza como nodo indefinido con nombre **expS**. Se señala en la pila, de manera que cualquier árbol parse de nodos creados y almacenados en la pila por no-terminales en la expansión de **ExpSuma()** será sacado de la misma y pasarán a formar parte de la estructura del nodo **expS** como hijos de éste. Los hijos generados tendrán como nombre en la pila **ExpMult**.

Si se quiere suprimir la creación de un nodo por una producción, se puede usar la sintaxis siguiente:

```
void ExpSuma() #void : {}
{
        ExpMult() ( ( "+" | "-" ) ExpMult() )*
}
```

Ahora cualquier nodo del árbol parser almacenado en la pila que ha sido generado por el no-terminal **ExpSuma()**, permanecerá en la pila para ser sacado de la misma y convertirse en hijo de una producción ascendente en el árbol, pero no pasará a formar parte de **ExpSuma()**, ya que este nodo no está en la pila. Sus hijos (producciones) pasarán a ser los hijos del nodo inmediatamente superior a **ExpSuma()**. Esta operación se puede hacer por defecto para los nodos no decorados (sin especificación de atributos) usando la opción **NODE_DEFAULT_VOID**.

Esto se consigue añadiendo, ala zona de opciones de **ejemplo1bis.jjt**, el siguiente código:

```
options {
        MULTI=true;
        NODE_DEFAULT_VOID=true;
}
```

El efecto que tiene esta opción sobre el código es el "ahorro" de indicar cada no-terminal como **#void**:

```
void ExpSuma() : {}
{
        ExpMult() ( ( "+" | "-" ) ExpMult() )*
}
```

Una forma de optimización de código utilizando el constructor # consiste en lo siguiente:

```
void ExpSuma() #void: {}
{
        (ExpMult() ( ( "+" | "-" ) ExpMult() )*)#Suma(>1)
}
```

El nodo **ExpSuma()** tendrá como hijos a uno o varios nodos llamados **Suma**.

El constructor **#Suma** es de tipo condicional, e indica que es de aridad mayor que 1. Se utiliza el tipo condicional ya que no se sabe cuántos no terminales de este tipo se han de generar para reconocer la sentencia que le paso a la gramática. Ejerce de operador en forma postfija, y su ámbito se corresponde con la unidad de expansión que le precede inmediatamente.

5.6.3 Ámbitos de Nodos y Acciones de Usuario

Cada nodo está asociado a un ámbito. Para poder acceder al nodo, las acciones que el usuario defina para dicho acceso han de tener el mismo ámbito que el nodo, y además se ha de usar el método especial **jjThis** para referirse al mismo. Este identificador se declara implícitamente de la siguiente manera:

```
SimpleNode Inicio() : {}
{
        Expresion() ";"
        { return jjtThis; }
}
```

De esta forma es más fácil el acceso a los campos y métodos del nodo al que se quiere acceder. Este identificador devuelve siempre el ámbito al que pertenece el nodo.

Un ámbito es la unidad que se expande y que le precede de forma inmediata al nodo con especificación de atributos (*decorated node*); por ejemplo, puede ser una expresión entre paréntesis. Cuando la signatura de la producción se decora (puede tener al nodo por defecto de forma implícita), el ámbito es la parte derecha de la producción, incluyendo el bloque de la declaración.

También se puede usar el método **jjThis** pasándole como parámetro la parte izquierda de una expansión de referencia. Un ejemplo basado en la gramática que estoy usando es el siguiente:

```
void ExpUnaria() : {}
{
        "(" Expresion() ")" | Identificador() | <ENTERO>
}

void Identificador() #Identificador: {
        Token token;}
{
        token=<IDENTIFICADOR> { jjtThis.setName(token.image); }
}
```

Aquí, **jjtThis** "trabaja" en el ámbito del nodo en el que ha sido invocado. Es decir, las modificaciones que se hagan sólo tendrán efecto en este nodo. Esta operación se puede realizar mediante el uso de la opción **NODE_SCOPE_HOOKS**, que, por defecto, está evaluada a false (ver la sección *Node Scope Hooks* para más información). Al sacar por pantalla los nodos almacenados en la pila observamos que la salida correspondiente a la entrada **(a + b)*(c + d)** es:

```
Inicio
Mult
        Suma
                Identificador: a
                Identificador: b
        Suma
                Identificador: c
                Identificador: d
```

Para poder actualizar el valor del campo nombre de cada nodo, hemos de implementar nosotros mismos el método **setName()** (o cualquier otro del que queramos hacer uso, como **setFirstToken()**, **setLastToken()**, **getName()**...). La forma y el lugar correcto de implementación se explican en la zona *Node Scope Hooks* del presente documento. La acción final que realiza un usuario en el ámbito de un nodo es distinta de la de todos los demás. Cuando el código se ejecuta, los hijos del nodo son sacados de la pila y añadidos a la estructura del nodo, el cual se almacena por sí mismo en la pila. Se puede acceder a los hijos del padre mediante el método **jjGetChild ()**. Estos métodos de acceso a los atributos de cada nodo se han implementado de forma automática, como más arriba se explica.

Otras acciones de usuario sólo pueden acceder a los hijos en la pila, debido a que aún no son parte del padre. No sirven los métodos que impliquen accesos al nodo (tales como **jjGetChild ()**).

Un nodo condicional que tenga un descriptor de nodo y sea evaluado a false, no será añadido a la pila, ni se le añadirá ningún hijo a su estructura. El usuario final, dentro del ámbito de un nodo condicional, puede precisar si el nodo ha sido creado o no mediante la llamada al método **nodeCreated** () (implementado en **SimpleNode.java**).Devuelve true si la condición del nodo se satisface, si el nodo ha sido creado y si ha sido almacenado en la pila. Devuelve false en otro caso.

5.6.4 Manejo de Excepciones

Una excepción lanzada por una expansión en el ámbito de un nodo que no es capturada en dicho ámbito, es capturada por JJTree. Cuando esto ocurre, cualquier nodo que haya sido almacenado en la pila es eliminado junto con su ámbito correspondiente. Entonces, la excepción se vuelve a recoger.

Este tipo de manejo de las excepciones tiene por objetivo hacer posible una implementación de recuperación de errores y continuar con la generación del parser a partir de un estado conocido.

Es importante hacer notar que, actualmente, JJTree no detecta cuándo una excepción es lanzada a partir de una acción de usuario dentro del ámbito de un nodo, con lo que, probablemente, una excepción puede ser tratada de forma incorrecta.

5.6.5 *Node Scope Hooks*

Esta opción altera el comportamiento de JJTree, de manera que, automáticamente, todos los tokens reconocidos son capturados. Si la opción **NODE_SCOPE_HOOK** está evaluada a true, JJTree genera llamadas a dos métodos definidos por el usuario en la entrada y salida de cada ámbito de nodo. Dichos métodos son:

```
void jjtreeOpenNodeScope(Node nodo)
void jjtreeCloseNodeScope(Node nodo)
```

Estos métodos han de ser definidos de manera que tengan acceso al primer y último token del nodo generado. De esta manera, se puede iterar desde el primer token encontrado hasta el último, sin miedo a que cambie el ámbito de la iteración, debido a que dichos métodos permanecen en el ámbito del nodo actual.

Se han de añadir los campos siguientes: **Token primero**, **ultimo**;

Se han de implementar en el fichero generado por javacc de forma automática llamado **SimpleNode.jav**a, ya que de esta clase, javac se alimenta de todos los métodos usados en el fuente con extensión **.jjt**.

El fichero **SimpleNode.java** del **ejemplo3.jjt** se presenta a continuación como apoyo a la explicación dada anteriormente:

```
public class SimpleNode implements Node {
    protected Node parent;
    protected Node[] children;
    protected int id;
    protected String name;
    protected Token token,first,last;
    protected ejemplo3 parser;

    public SimpleNode(int i) {
        id = i;
    }
    public SimpleNode(ejemplo3 p, int i) {
        this(i);
        parser = p;
    }

    public void jjtOpen() {
    }
    public void jjtClose() {
    }
    public void jjtSetParent(Node n) { parent = n; }
    public Node jjtGetParent() { return parent; }
    public void jjtAddChild(Node n, int i) {
        if (children == null) {
            children = new Node[i + 1];
        } else if (i >= children.length) {
            Node c[] = new Node[i + 1];
            System.arraycopy(children, 0, c, 0, children.length);
```

```
                children = c;
            }
            children[i] = n;
        }
        public Node jjtGetChild(int i) {
            return children[i];
        }
        public int jjtGetNumChildren() {
            return (children == null) ? 0 : children.length;
        }
        /* You can override these two methods in subclasses of SimpleNode to
        customize the way the node appears when the tree is dumped.  If
        your output uses more than one line you should override
        toString(String), otherwise overriding toString() is probably all
        you need to do. */
        public void setName(String s) {
            this.name=s;
        }
        public String getName() {
            return this.name;
        }
        public void setFirstToken(Token t) {
            this.first=t;
        }
        public Token getFirstToken() {
            return this.first;
        }
        public void setLastToken(Token t) {
            this.last=t;
        }
        public Token getLastToken() {
            return this.last;
        }
        public String toString() {
            return ejemplo3TreeConstants.jjtNodeName[id]+":"+name;
         }
         public String toString(String prefix) { return prefix + toString();
         }
        /* Override this method if you want to customize how the node dumps
        out its children. */
        public void dump(String prefix) {
            System.out.println(toString(prefix));
            if (children != null) {
                for (int i = 0; i < children.length; ++i) {
                    SimpleNode n = (SimpleNode)children[i];
                    if (n != null) {
                        n.dump(prefix + " ");
                    }
                }
```

```
                }
        }
}
```

Aquellas variables y métodos que se presentan en negrita son los que necesitamos definir (o redefinir) para que javac reconozca todos aquellos métodos que hemos definido en el fichero principal .jjt

El resto de los métodos son generados automáticamente por javacc, pudiendo modificarlos siempre y cuando seamos conscientes de las modificaciones llevadas a cabo. Cualquier acceso a un nodo o cualquier operación incorrecta llevará a la generación de errores. Si no los modificamos, tenemos la seguridad de que dichos métodos realizan las acciones que les son asignadas por defecto.Otro uso que se le puede dar es almacenar el parser objeto en el nodo, de manera que se puede proporcionar el estado que debe ser compartido por todos los nodos generados por el parser. Por ejemplo, el parser podría mantener una tabla de símbolos.

5.6.6 El ciclo de vida de un nodo

Un nodo pasa por una secuencia determinada de pasos en su generación. La secuencia vista desde el propio nodo es la siguiente:

1. El constructor del nodo se llama con un único parámetro de tipo entero. Dicho parámetro nos dice qué tipo de nodo es, siendo esto especialmente útil en modo simple. JJTree genera automáticamente un fichero llamado parserTreeConstants.java y declara constantes en Java para el identificador del nodo. Además, declara un array de Strings llamado jjtNodeName [], el cual asigna identificadores enteros a los nombres de los nodos. Para nuestro ejemplo2, el fichero es el siguiente:

```
public interface ejemplo2TreeConstants{
    public int JJTINICIO = 0;
    public int JJTVOID = 1;
    public int JJTSUMA = 2;
    public int JJTMULT = 3;
    public int JJTIDENTIFICADOR = 4;
    public int JJTENTERO = 5;
    public String[] jjtNodeName = {
        "Inicio",
        "void",
        "Suma",
        "Mult",
        "Identificador",
        "Entero",
      };
    }
```

2. Se realiza una llamada al método jjtOpen ().
3. Si la opción **NODE_SCOPE_HOOK** está activa, se llama al método

parser definido por el usuario **openNodeScope ()** y se le pasa el nodo como parámetro. Por ejemplo, podría almacenar el primer token en el nodo:

```
static void jjtreeOpenNodeScope(Node nodo) {
        ((SimpleNode)nodo).setFirstToken(getToken(1));
}
```

Este método ha de implementarse en la clase principal, es decir, en el **ejemplo2.jjt**

4. Si una excepción sin manejador se lanza mientras que el nodo se está generando, dicho nodo se descarta. JJTree nunca volverá a hacer referencia a este nodo. Dicho nodo no se cerrará, y en el método de usuario **closeNodeHook()** no utilizará dicho nodo como parámetro en una llamada.
5. En otro caso, si el nodo es condicional, y su condición se evalúa a false, el nodo se descarta. No se cerrará dicho nodo, a no ser que el método definido por el usuario **closeNodeHook ()** pueda utilizarse en una llamada con este nodo como parámetro.
6. En otro caso, el número de hijos del nodo especificado en la expresión entera del nodo definido, o todos los nodos que fueron almacenados en la pila en un ámbito de nodo condicional, son añadidos al nodo (dicho número es la aridad del nodo; si es condicional, se define **#Id(>1))**. El orden en el que se añaden no se especifica.
7. Se llama al método **jjtClose ()**.
8. El nodo es almacenado en la pila.
9. Si la opción **NODE_SCOPE_HOOK** está activa, el método parser definido por el usuario **closeNodeScope ()** se llama y utiliza el nodo como parámetro.
10. Si el nodo no es el nodo raíz, se añade como estructura de hijo a otro nodo y se llama a su método **jjtParent ()**.

5.6.7 Visitor Support

JJTree provee algunos soportes básicos para el patrón de diseño visitor. Si la opción **VISITOR** está a true, JJTree añadirá un método **jjtAccept ()** en todas las clases de nodo que genere, además de la creación de una interfaz de usuario que pueda ser implementada y pasada a los nodos.

El nombre de dicha interfaz se construye mediante la adición **Visitor** al nombre del parser. La interfaz se genera cada vez que JJTree se ejecuta, de manera que representa de forma exacta el conjunto de nodos usados por el parser. Si la clase implementada no ha sido actualizada para los nodos nuevos generados, producirá

errores en tiempo de compilación.

5.6.7.1 Parámetros de Ejecución

JJTree proporciona opciones para la línea de comando y para las opciones de declaraciones de JavaCC:

1. **BUILD_NODE_FILES** (**true** por defecto): genera implementaciones de muestra para SimpleNode y para cualesquiera otros nodos usados en la gramática.
2. **MULTI** (**false** por defecto): genera el árbol parse en modo multi. Cada nodo (no-terminal) creado y no declarado como **void**, genera un fichero de nombre **nombreNoTerminal.java**, en el cual cada nodo extiende de **SimpleNode**. Estos ficheros se generan de forma automática.
3. **NODE_DEFAULT_VOID** (**false** por defecto): en vez de crear cada producción no-decorada como un nodo indefinido, lo crea void en su lugar. Cualquier nodo decorado no se verá afectado por esta opción.
4. **NODE_FACTORY** (**false** por defecto): usa un método de generación para construir nodos con la siguiente estructura:
   ```
   public static Node jjtCreate (int id)
   ```
5. **NODE_PACKAGE** (**""** por defecto): el paquete para generar el nodo en una clase. Por defecto, contiene el paquete del parser.
6. **NODE_PREFIX** (**"AST"** por defecto): el prefijo usado para construir los nombres de clase del nodo a partir de los identificadores del nodo en modo multi. Es el prefijo que se pone automáticamente por defecto cuando se generan los ficheros que contienen las implementaciones de los nodos que extienden de **SimpleNode**.
7. **NODE_SCOPE_HOOK** (**false** por defecto): inserta llamadas a los métodos parser definidos por el usuario en cada entrada y salida de cada ámbito de nodo. Ver Node Scope Hooks
8. **NODE_USERS_PARSER** (**false** por defecto): JJTree usará una forma alternativa de las rutinas de construcción de nodos donde se le pasa el parser del objeto. Por ejemplo:
   ```
   public static Node MiNodo.jjtCreate(MiParser p, int id);
                MiNodo (MiParser p, int id);
   ```
9. **STATIC** (**true** por defecto): genera código para un parser static. Debe ser usado de forma consecuente con las opciones equivalentes proporcionadas por JavaCC. El valor de esta opción se muestra en el fuente de JavaCC.
10. **VISITOR** (**false** por defecto): inserta un método jjtAccept () en las clases del nodo, y genera una implementación de visitor con una entrada para

cada tipo de nodo usado en la gramática.

11. **VISITOR_EXCEPTION** (**""** por defecto): si esta opción está activa, se usa en las estructuras de los métodos **jjtAccept ()** y **visit ()**. Mucho cuidado porque esta opción se eliminará en versiones posteriores de JJTree.

5.6.8 Estados en JTree

JJTree mantiene su estado en un campo de la clase del parser llamado jjtree. Se puede usar métodos en este miembro para manipular la pila del nodo. Algunos de los métodos generados automáticamente para esta clase están expuestos más arriba. También se indica en qué fichero se encuentra almacenado el código generado.

5.6.8.1 Objetos Node

Todos los nodos AST (*Abstract Syntax Tree*) deben implementar esta interfaz. Proporciona un motor básico para construir relaciones padre-hijos entre nodos.

Esta interfaz es común para todos los archivos **.jjt** que se lancen mediante el ejecutable **jjtree**. El contenido es:

```
public interface Node {
        /* Este método se llama después de que el nodo haya pasado a ser el nodo
        actual. Indica que se le pueden añadir los nodos hijo*/
        public void jjtOpen ();

        /* Este método se llama después de haber añadido todos los nodos hijo*/
        public void jjtClose ();

        /* Los siguientes 2 métodos se usan para asignar/devolver el padre de un
        nodo*/
        public void jjtSetParent (Node n);
        public Node jjtGetParent () ;

        /*Este método añade su argumento a la lista de nodos hijo*/
        public void jjtAddChild (Node n, int i);

        /*Este método devuelve un nodo hijo. Se numeran desde 0, de izquierda a
        derecha*/
        public Node jjtGetChild (int i);

        /*Devuelve el número de hijos que tiene el nodo*/
        int jjtGetNumChildren ();
}
```

La clase **SimpleNode** implementa la interfaz **Node**, y si no existe, se genera de forma automática por JJTree. Se puede usar esta clase como una plantilla, como una superclase para las implementaciones de nodos, o se puede modificar a conveniencia.

SimpleNode provee, de forma adicional, un mecanismo simple de "recolector"

recursivo del nodo y sus hijos. Dicho método es el que aparece en los ejemplos arriba descritos como **dump()**.

El parámetro de tipo **String** del método **dump()** se usa como prefijo para indicar la estructura jerárquica del árbol.

Si la opción **VISITOR** está activa, se genera otro método que puede ser de utilidad:

```
public void childrenAccept (MyParserVisitor visitor);
```

Este método recorre los nodos hijos en orden, invitándoles a aceptar a visitor. Este método es muy útil cuando se ha implementado una estructura en preorden o postorden.

5.6.8.2 Pasos de Ejecución

La ejecución de los ficheros **.bat** han de lanzarse en el editor de comandos de MS-DOS.

1. Si el fichero a compilar tiene extensión **.jj**:

```
javacc archivo.jj
javac archivo.java
java archivo
```

2. Si el fichero tiene extensión **.jjt**:

```
jjtree archivo.jjt
javacc archivo.jj
javac archivo.java
java archivo
```

Los archivos generados al ejecutar **jjtree <archivo.jjt>** son:

archivoTreeConstants.java
JJTarchivoState.java
Node.java
SimpleNode.java

Si el fuente está en modo multi, se generará un fichero **.java** por cada no-terminal no declarado como **void**, es decir, que esté decorado. Los nombres de los ficheros serán los mismos que los nombres de los no-terminales.

Los archivos generados al ejecutar **javacc <archivo.jj>** son:

archivo.java
archivoConstants.java
archivoTokenManager.java
TokenMgrError.java
ParseException.java
Token.java
SimpleCharStream.java

El siguiente paso es la modificación del fichero **SimpleNode.java** con los métodos que nos sean de utilidad, esto es, que hemos invocado en el fuente **.jjt**.

De esta manera, no se generarán errores tipo "*method not found*" o "*symbol not found*" a la hora de compilar el fichero **.java** generado automáticamente por javacc.

Hemos de ejecutar el comando javac en el directorio contenedor o, si queremos ahorrarnos tiempo, actualizar el *path* con la dirección en la que se encuentra **javac.bat** y **java.bat**

La linea de comandos a ejecutar, en el caso de encontrarnos en el director contenedor de la javac es:

```
C:\jdk1.5.0_01\bin> javac -classpath <camino de los ficheros fuente> ejemplo.java
C:\jdk1.5.0_01\bin> java -classpath <camino de los ficheros fuente> ejemplo
```

Nos pedirá la introducción de la sentencia(s) a reconocer, si la entrada seleccionada es la estándar, o pasarle el fichero en el que se encuentra el código a reconocer mediante el comando

```
C:\jdk1.5.0_01\bin> java -classpath <path de los ficheros fuente> ejemplo > fichero
```

5.6.9 Manual de Referencia de JJDoc

JJDoc toma la especificación del parser de JavaCC y produce la correspondiente documentación para la gramática BNF. Puede funcionar de tres formas distintas, dependiendo del comando introducido en la zona de opciones:

- **TEXT**: Por defecto, esta opción está a false.Inicializando esta opción a true, JJDoc genera una descripción simple en formato texto de la gramática en BNF. Algunos formateos se hacen vía caracteres tabuladores, pero el objetivo es dejarlo tan simple como sea posible.
- **ONE_TABLE**: Por defecto, true.El valor por defecto de ONE_TABLE se usa para generar una única tabla HTML para la notación BNF. Actualizar el valor a false supone la generación de una tabla por cada producción gramatical.
- **OUTPUT_FILE**: El funcionamiento por defecto es poner la salida de JJDoc en un fichero junto con la extensión **.html** o **.txt** añadida como sufijo al nombre del fichero de entrada. Puede suministrar un nombre de fichero distinto con esta opción.

Cuando está activada la opción **TEXT** o está puesta a false la opción **ONE_TABLE**, los comentarios en el fuente JavaCC que preceden inmediatamente a una producción son obviados por la documentación generada.

Para **ONE_TABLE**, obtenemos un fichero **ejemplo2.html** cuyo contenido es:

BNF for ejemplo2.jj (NON-TERMINALS)

Inicio	::=	Expresion ";"
Expresion	::=	ExpSuma
ExpSuma	::=	(ExpMult (("+" | "-") ExpMult)*)

ExpMult	::=	(ExpUnaria (("*" | "/" | "%") ExpUnaria)*)
ExpUnaria	::=	"(" Expresion ")"
	|	Identificador
	|	Entero
Identificador	::=	(<IDENTIFICADOR>)
Entero	::=	<ENTERO>

Para **TEXT=true**, obtenemos un fichero **ejemplo2.txt** cuyo contenido es:

```
DOCUMENT START
NON-TERMINALS
        Inicio      :=      Expresion ";"
        Expresion   :=      ExpSuma
        ExpSuma     :=      ( ExpMult ( ( "+" | "-" ) ExpMult )* )
        ExpMult :=  ( ExpUnaria ( ( "*" | "/" | "%" ) ExpUnaria )* )
        ExpUnaria   :=      "(" Expresion ")"
                |           Identificador
                |           Entero
        Identificador   :=  ( <IDENTIFICADOR> )
        Entero  :=      <ENTERO>
DOCUMENT END
```