Capítulo 2

INTRODUCCIÓN A JAVACC.

2.1. INTRODUCCIÓN

Como se comentó en el capitulo anterior, la herramienta que vamos a usar para definir nuestro lenguaje, y por consiguiente su intérprete, va a ser el metacompilador JavaCC.

Un metacompilador o generador de parsers es una herramienta que, a partir de la especificación de un lenguaje, construye un programa o analizador que es capaz de reconocer secuencias o elementos de dicho lenguaje. En general, la especificación del lenguaje abarca tanto el aspecto léxico como el sintáctico, y son los que permiten la construcción del parser, mientras que el aspecto semántico del lenguaje se deja en manos del usuario, para que lo ensamble una vez obtenido el parser.

En este capítulo vamos a desarrollar una introducción a JavaCC a modo de breve guía. La idea es hacer que el lector aprenda las nociones básicas de uso y funcionamiento de manera que este capítulo pueda ser usado como un manual de apoyo básico para el desarrollo de compiladores mediante JavaCC. No se va a entrar en

profundizaciones demasiado complejas, y si algún lector desea aprender características mayores sobre esta herramienta se recomienda que use la bibliografía referente a JavaCC.

2.2. CARACTERÍSTICAS DE JAVACC

JavaCC integra las funciones de análisis léxico y análisis sintáctico en una sola herramienta, obteniendo a la salida código java –a diferencia de *lex/yacc* cuya salida es código *C*-.

Antes que nada veamos un ejemplo sobre el cual se va a ir haciendo un seguimiento de cada uno de los distintos puntos que vayamos comentando. En él se muestra una gramática reconocedora de una calculadora. Más adelante, en este mismo capítulo, lo modificaremos con el objetivo de mostrar mayores funcionalidades que aporta JavaCC.

```
options {
        LOOKAHEAD=1;
}
PARSER_BEGIN(Calculadora)
        public class Calculadora {
                public static void main(String args[]) throws ParseException {
                        Calc1 parser = new Calc1(System.in);
                        while (true) {
                                System.out.print("Introduzca una expresion: ");
                                System.out.flush();
                                 try {
                                        switch (parser.one_line()) {
                                                case -1:
                                                System.exit(0);
                                                default:
                                                break;
                                        }
                                }
                                catch (ParseException x) {
                                        System.out.println("Finalizando.");
                                        throw x;
                                }
                        }
                }
        }
PARSER_END(Calculadora)
```

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

TOKEN :
{
        < EOL: "\n" >
}

TOKEN : /* OPERADORES */
{
        < MAS: "+" >
|       < MENOS: "-" >
|       < POR: "*" >
|       < ENTRE: "/" >
}

TOKEN :
{
   < CONSTANTE: ( <DIGITO> )+ >
|  < #DIGITO: ["0" - "9"] >
}

int one_line() :
{}
{
        suma() <EOL>{ return 1; }
        | <EOL>{ return 0; }
        | <EOF>{ return -1; }
}

void suma() :
{ }
{
        termino() (( <MAS> | <MENOS> ) termino())*
}

void termino() :
{ }
{
        unario() (( <POR> | <ENTRE> ) unario())*
}
void unario() :
{ }
{
            <MENOS> elemento()
        |   elemento()
}

void elemento() :
{}
{
           <CONSTANTE>
        |  "(" suma() ")"
}
```

Ejemplo 2.1. Gramática reconocedora de una Calculadora.

La principal diferencia con *lex/yacc* es que los analizadores sintácticos generados por *JavaCC* son analizadores sintácticos descendentes de tipo LL(k), mientras que los analizadores sintácticos generados por *yacc* son de tipo ascendente LALR. Otra diferencia bastante importante es que las especificaciones léxicas pueden estar incluidas dentro de la especificación de la gramática. Por ejemplo, podemos escribir algo como esto:

```
sentenciaIf : {} { "if" "(" expresion() ")" sentencia() }
```

Ejemplo 2.2.

y automáticamente se ocupa de generar *tokens* para "**if**" y para "**(**".

También es necesario reseñar que el análisis léxico en JavaCC genera combinaciones de autómatas finitos deterministas (AFD) y autómatas finitos no deterministas(AFND); por contra, *lex* siempre genera autómatas finitos deterministas.

En definitiva diremos que Java Compiler Compiler (JavaCC) es un potente generador de parsers descendentes escritos en lenguaje Java puro. Es, quizá, uno de los generadores de parsers más populares dada su multiplataformidad. Es por esto que se recomienda su uso para usuarios más afines a los lenguajes de programación orientados a objetos.

Dentro de las características de este metacompilador hay algunas que son aportadas por *lex/yacc* y otras que no. Principalmente destacamos:

- Análisis descendente, permitiendo el uso de gramáticas mas generales y la posibilidad de poder utilizar atributos en el árbol sintáctico durante el parsing.
- Especificaciones léxicas y gramaticales en un solo archivo. De esta manera la gramática puede ser leída y mantenida más fácilmente gracias al uso de las expresiones regulares dentro de la gramática.
- Permite extender especificaciones BNF mediante la utilización de expresiones regulares, tales como (A)*, (A)+.

- Ofrece estados léxicos y la capacidad de agregar acciones léxicas incluyendo un bloque de código java tras el identificador de un token. Además de los conceptos de token, more, skip, cambio de estados, etc. Ello permite trabajar con especificaciones más claras, a la vez que permite un mejor manejo de mensajes de error y advertencias de JavaCC.
- Genera por defecto un parser LL(1). sin embargo, puede haber porciones de la gramática que no son LL(1). JavaCC ofrece la posibilidad de resolver las ambigüedades shift-shift localmente al punto del conflicto. En otras palabras, permite que el parser se vuelva LL(k) sólo en tales puntos; pero se conserva LL(1) en el resto de las producciones para obtener una mejor actuación.
- Permite el uso de Tokens Especiales que son ignorados por el parser; pero están disponibles para poder ser procesados por el desarrollador.
- Las especificaciones léxicas pueden definir tokens de manera tal que a nivel global no se diferencien las mayúsculas de las minúsculas en la especificación léxica completa o en una especificación léxica individual
- El análisis del parsing y los pasos de procesamiento de tokens pueden realizarse en profundidad por medio de la utilización de las opciones DEBUG_PARSER, DEBUG_LOOKAHEAD, y DEBUG_TOKEN_MANAGER,.
- JavaCC incluye JJTree, un preprocesador para el desarrollo de árboles, con características muy poderosas.
- De entre los generadores de parsers, JavaCC se halla entre los que tienen mejor manejo de errores. Los parsers generados por JavaCC son capaces de localizar exactamente la ubicación de los errores, proporcionando información diagnóstica completa.
- JavaCC incluye una herramienta llamada JJDoc que convierte los archivos de la gramática en archivos de documentación.

- El analizador léxico de JavaCC puede manejar entradas Unicode, y las especificaciones léxicas también pueden incluir cualquier caracter Unicode. Esto facilita la descripción de los elementos del lenguaje, tales como los identificadores Java que permiten ciertos caracteres Unicode que no son ASCII, pero no otros.

- JavaCC es quizá el generador de parsers usado con aplicaciones Java más popular.
- JavaCC ofrece muchas opciones diferentes para personalizar su comportamiento y el comportamiento de los parsers generados.

2.3. ESTRUCTURA GENERAL DE UN PROGRAMA EN JAVACC

Cualquier código escrito para JavaCC obedece a la siguiente estructura:

```
Opciones_javacc
"PARSER_BEGIN" "("<IDENTIFICADOR>")"
Unidad de compilación java
"PARSER_END" "("<IDENTIFICADOR>")"
(reglas de produccion)*
<EOF>
```

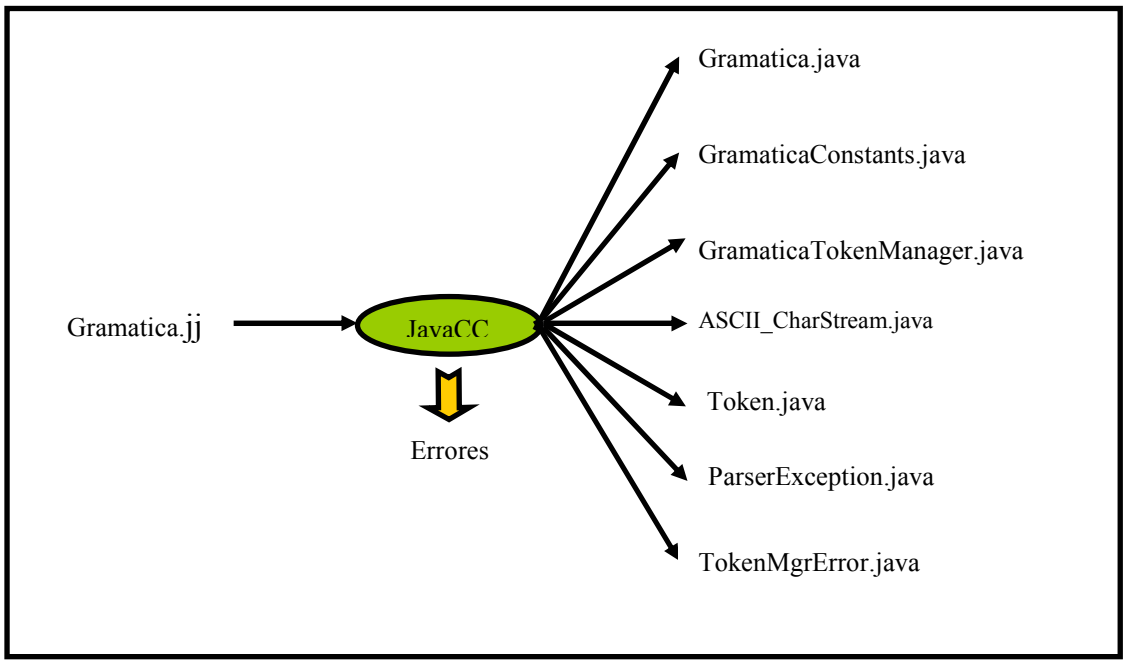
Esquema 2.1. Estructura del código.

El fichero de entrada comienza con una lista de opciones, la cual es opcional. Seguidamente nos encontramos la unidad de compilación *java* la cual se encuentra encerrada entre *PARSER_BEGIN(nombre)* y *PARSER_END(nombre)*. Después nos encontramos con una lista de reglas de producción (cada una de estas partes son claramente distinguibles en el ejemplo 2.1.).

El *nombre* que sigue a *PARSER_BEGIN* y a *PARSER_END* debe ser el mismo, y éste identifica al analizador sintáctico que va a ser generado con posterioridad. Por ejemplo si nuestro *nombre* es Gramática, se generan los siguientes archivos:

- **Gramatica.java**: El analizador sintáctico.
- **GramaticaTokenManager.java**: gestor de *tokens* para el analizador léxico.
- **GramaticaConstants.java**: El analizador léxico no maneja los *tokens* por el nombre que especificamos nosotros en el código, sino que a cada uno de ellos le asigna un número. En este archivo se encuentra cada uno de los tokens junto con el número que le ha sido asignado por el analizador léxico.

Además de éstos, también se generan otros ficheros (Token.java y ParseException.java, ASCII_CharStream.java, TokenMgrError.java), pero estos ficheros son los mismos para cualquier gramática y pueden conservarse de una para otra.



Los archivos Token.java , ParseException.java, ASCII_CharStream.java y TokenMgrError.java son los mismos para cualquier gramática y pueden conservarse de una para otra.

Figura 2.1: Archivos generados por *JavaCC*

Entre *PARSER_BEGIN* y *PARSER_END*, se encuentra una unidad de compilación *Java* (el contenido completo de un archivo escrito en Java). Ésta puede ser cualquier código *Java* siempre que contenga una declaración de clase cuyo nombre sea el mismo que el analizador sintáctico que va a ser generado.(Gramática en el ejemplo anterior). Esta parte del archivo de gramática tiene esta forma:

```
PARSER_BEGIN(nombre)
. . .
class nombre. . . . . {
. . .
}
. . .
PARSER_END(nombre)
```

Esquema 2.2. Estructura de la unidad de compilación java.

JavaCC no realiza un chequeo minucioso de la unidad de compilación por lo que es posible que un fichero de gramática pase el test de *JavaCC* y genere los archivos correspondientes y luego produzca errores al ser compilados.

El archivo correspondiente al analizador sintáctico que es generado por *JavaCC* contiene la unidad de compilación Java en su totalidad, además del código generado por el analizador sintáctico, el cual se incluye al final de su clase. En el caso del ejemplo 2.1. tendría la siguiente forma:

```
. . .
class Calculadora. . . . . {
. . .
// el código del analizador sintáctico es insertado aquí
}
. . .
```

Esquema 2.3. Estructura de la clase generada.

El código del analizador sintáctico incluye la declaración de un método público correspondiente a cada no terminal de la gramática. La realización del análisis sintáctico a partir de un no terminal se consigue invocando al método correspondiente a ese no terminal.[1]

2.4. OPCIONES

La sección *options*, si está presente, comienza con la palabra reservada "***options***" seguida de una lista de una o más opciones entre llaves. La misma opción no debe establecerse más de una vez. En el ejemplo 2.1. sólo se ha usado una de las opciones (LOOKAHEAD=1).

Las opciones pueden ser especificadas tanto en el archivo de la gramática como en la línea de comandos. En caso de estar presentes en el archivo , éstas deben aparecer al principio precedidas por la palabra "options". Y en el caso en que se especifiquen en la línea de comandos, éstas tienen mayor prioridad que las especificadas en el código.

[1] Para ver al completo las Reglas de Producción de especificaciones JavaCC consulte el apéndice respectivo.

opciones::= [“options” “{“ (valores_de_las_opciones)* “}”]

Esquema 2.4. Estructura de la zona de opciones.

Los nombres de las opciones van en mayúsculas.

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 parser antes de tomar una decisión de parsing. El valor por defecto es 1, con lo cual se trabaja con parser LL(1), aunque el número puede variar para alguna/s producción/es en particular. Cuanto más pequeño sea el número de tokens del lookahead, más veloz será el parser.
  Valor por defecto: 1
- **CHOICE_AMBIGUITY_CHECK:** Es el número de tokens que se consideran en las elecciones de chequeo por ambigüedad, de la forma "A | B | ...". Por ejemplo: si existen dos tokens prefijos comunes para A y B, pero no tres; entonces por medio de esta opción, JavaCC permite usar un lookahead de 3 con propósitos de desambiguación. La utilización de esta opción provee mayor información y flexibilidad ante las ambigüedades, pero a costo de un mayor tiempo de procesamiento.
  Valor por defecto: 2
- **STATIC:** Si el valor es true, todos los métodos y clases variables se especifican como estáticas en el parser generado y en el token manager. Esto permite la presencia de un único objeto parser, pero mejora la actuación del procedimiento de parsing. Para ejecutar múltiples parsers durante la ejecución de un programa Java, debe invocarse el método ReInit() para reinicializar el parser, si es que había sido definido como estático. Si el parser no es estático, puede utilizarse el operador "*new*" para construir los parsers que sean necesarios; los mismos, podrían ser usados simultáneamente en threads diferentes.
  Valor por defecto: true

- **DEBUG_PARSER:** Esta opción se utiliza para obtener información de debugging desde el parser generado. Si el valor de la opción es true, el parser genera un trace de sus acciones. Dichos traces pueden deshabilitarse por medio de la invocación del método disable_tracing() en la clase del parser generado, y, recíprocamente, pueden habilitarse con el método enable_tracing().
  Valor por defecto: false
- **IGNORE_CASE:** Si su valor es true, el token manager generado no efectúa diferencia entre letras mayúsculas y minúsculas en las especificaciones de tokens y los archivos de entrada. Es muy usado para escribir gramáticas para lenguajes como HTML.
  Valor por defecto: false.
- **FORCE_LA_CHECK:** Esta opción determina los controles de chequeo de ambigüedad de lookahead ejecutados por JavaCC. Cuando el valor es false, el chequeo de ambigüedad del lookahead se ejecuta para todos los puntos de elección donde se usa el lookahead de 1 por defecto. El chequeo de ambigüedad de lookahead no se ejecuta en los puntos donde existe una especificación explícita de lookahead, o si la opción LOOKAHEAD tiene un valor distinto de 1.
  Valor por defecto: false.
- **COMMON_TOKEN_ACTION:** Si el valor de esta opción es true, todas las invocaciones del método "*getNextToken()*" del token manager, causan una llamada al método definido "*CommonTokenAction()*" después de que el token haya sido leído por el token manager. El usuario debe definir este método en la sección TOKEN_MGR_DECLS. La signatura de este método es:

  void CommonTokenAction(Token t)

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

2.5. REGLAS DE PRODUCCIÓN

Existen cuatro tipos de reglas de producción en JavaCC: **las reglas de producción javacode** y **BNF** se usan para definir la gramática a partir de la cual va a

generarse el analizador sintáctico. **Las Reglas de producción mediante expresiones regulares**, se usan para definir los *tokens* de la gramática – el *token manager* es generado a partir de esta información-. La sección **token manager decls** se usa para introducir declaraciones en el analizador sintáctico que se generará con posterioridad.

2.5.1. Reglas de producción Javacode

Las reglas de producción JAVACODE son una forma de escribir código Java para una regla de producción en lugar de las típicas expansiones EBNF. Esto suele ser útil cuando tenemos la necesidad de reconocer algo que no es de contexto libre o algo para lo cual es difícil escribir una gramática que lo reconozca. Para ello nos serviremos de los métodos que el propio parser nos proporciona con tal fin, a saber: getToken(), jj_concume_token(), getNextToken(),etc. A continuación se muestra un ejemplo de uso de JAVACODE. En el ejemplo 2.3., el no terminal "skipToMatchingBrace" consume tokens de la entrada hasta que encuentra una "}".

```
JAVACODE
void skipToMatchingBrace(){
        Token tok;
        int nesting=1;
        while(true){
                tok=getToken(1);
                if(tok.kind==LBRACE)
                        nesting++;
                if(tok.kind==RBRACE){
                        nesting--;
                        if(nesting==0)
                                break;
                }
                tok=getNextToken();
        }
}
```

Ejemplo 2.3. Regla de producción Javacode.

Cuando usamos este tipo de reglas de producción tenemos que tener muchísimo cuidado ya que es muy fácil especificar bastante más de lo que realmente se quiere.

Este tipo de reglas son problemáticas cuando las utilizamos en lugares donde el analizador sintáctico tiene que hacer una elección. Supongamos que la regla de producción anterior es referenciada desde ésta:

```
void NT():
{}
{
        skip_to_matching_brace()
|
        otra_regla_de_produccion()
}
```

Ejemplo 2.4. Uso inadecuado de una regla de producción Javacode

En este punto JavaCC no sabría elegir entre las dos opciones. En cambio si la regla de producción JAVACODE es usada en un lugar donde no hay que elegir entre diversas reglas no hay ningún problema. Por ejemplo:

```
void NT():
{}
{
        "{" skip_to_matching_brace()
|
        "("lista_de_parametros() ")"
}
```

Ejemplo 2.5. Uso correcto de una regla de producción Javacode

Cuando se utilizan reglas de producción *JAVACODE* en puntos donde hay que elegir entre varias reglas, *JavaCC* nos avisará con un mensaje constatando éste hecho. Entonces tendremos que cambiar de forma explícita el *LOOKAHEAD*.

2.5.2. Reglas de producción BNF

Las reglas de producción BNF son la forma estándar de especificar gramáticas en *JavaCC*. Para mostrar su estructura y funcionamiento tomemos una de ellas del ejemplo 2.1., la relativa a un unario. Sobre ella diferenciaremos cada uno de los puntos que a continuación se exponen:

```
void unario() :
{ }
{
        <MENOS> elemento()
    |   elemento()
}
```

Ejemplo 2.6. Regla de producción BNF.

Cada regla de producción BNF consta de una parte izquierda la cual consiste en la especificación de un no terminal. Las reglas de producción BNF definen este no terminal en función de expansiones BNF que se encuentran en la parte derecha. El no terminal se escribe exactamente de la misma forma en que se declara un método en Java, (en el ejemplo 2.6. "void unario():") . Ya que cada no terminal se traduce a un método en el analizador sintáctico, la forma de escribir el no terminal hace que esta asociación sea obvia. El nombre del no terminal es el nombre del método, y los parámetros y el valor de retorno son los medios para pasar valores arriba y abajo dentro del árbol sintáctico. – *parse tree*- (Esto se mostrará más adelante).

La parte derecha de una regla de producción BNF tiene dos partes: La primera parte es un conjunto de declaraciones y código *Java* (bloque *Java*). Este código es generado al principio del método relacionado con el no terminal. Por lo tanto, cada vez que el no terminal es usado en el proceso de análisis, este código es ejecutado. En el ejemplo 2.6. esta parte aparece vacía (más adelante se mostrará un ejemplo que hace uso de ella). La segunda parte está formada por las expansiones *BNF*.

```
bnf_production::= java_return_type java_identifier "(" java_parameter_list ")" ":"
                  java_block
                  "{" expansion_choices "}"
```

Esquema 2.5. Estructura de una regla de producción BNF.

Las opciones de expansión son descritas como una secuencia de unidades de expansión separadas por barras horizontales ("|").

```
expansion ::=(expansion_unit)*
```

Esquema 2.6. Estructura de una expansión BNF.

Una expansión es descrita como una secuencia de unidades de expansión. Por lo tanto, para que una expansión sea reconocida de forma correcta , necesitamos que cada una de las unidades de expansión que la forman sean reconocidas de forma correcta . Por ejemplo, la expansión *<MENOS> elemento()* está formada por dos unidades de

expansión – <MENOS> y elemento() -. Para que una entrada sea reconocida con éxito, debe comenzar por "-", y finalizar con una expresión que se ajuste a elemento().

Una de la mayores ventajas que aporta JavaCC es el hecho de que se puede hacer uso de bloques de código java incrustados en cada una de la expansiones BNF. Este aspecto se verá claramente en el apartado dedicado a la gestión de atributos. Por medio del empleo de código java, se nos permite la capacidad de realizar acciones semánticas que puede actuar sobre las entidades sintácticas que se han ido generando. Combinando esto con el uso de los parámetros de entrada y de retorno de los no terminales, se puede tener un control completo sobre el paso de valores, tanto hacia arriba como hacia abajo, en el árbol sintáctico.

2.5.3. Reglas de producción mediante expresiones regulares

Las expresiónes regulares se usan para definir entidades léxicas las cuales van a ser procesadas por el *token manager*. Una regla de producción de este tipo comienza con la especificación de los estados léxicos a los cuales es aplicable.

```
regular_expr_production::= [lista_de_estados_lexicos]
                           regexpr_kind[ "[" "IGNORE CASE" "]" ] ":"
                           "{" regexpr_spec ( "|" regexpr_spec )* "}"
```

Esquema 2.7. Estructura de una regla de producción mediante expresiones regulares.

Existe un estado léxico estándar llamado "*DEFAULT*". Si la lista de estados léxicos se omite, la expresión regular se aplica al estado "*DEFAULT*".

```
lista_de_estados_lexicos::= "<" "*" ">"
                          | "<" identificador_java ("," identificador_java)* ">"
```

Esquema 2.8. Estructura de una lista de estados léxicos.

La lista de estados léxicos describe el conjunto de estados léxicos, para los cuales la expresión regular es aplicable. Si ésta es "<" "*" ">", entonces la expresión regular se puede aplicar a todos los estados léxicos. En otro caso se aplicará a todos los estados léxicos cuyo identificador se encuentre en la lista de identificadores del interior de los paréntesis angulares.

A continuación se especifica el tipo de regla de producción. Existen cuatro tipos basadas en expresiones regulares:

- ***TOKEN***: La expresión regular de esta regla de producción describe tokens de la gramática. El *token manager* crea un objeto de tipo *Token* para cada una de las entradas que se ajusten a dicha expresión regular, y luego lo retorna al analizador sintáctico.
- ***SPECIAL_TOKEN***: La expresión regular de esta regla de producción describe *tokens* especiales. Los *tokens* especiales son iguales que los *tokens* sólo que no tienen importancia durante el análisis sintáctico. Son muy útiles cuando hay que procesar entidades léxicas, del tipo de comentarios, las cuales no tienen ninguna importancia para el análisis sintáctico, pero son una parte muy importante del fichero de entrada.
- ***SKIP***: Cualquier entrada que se ajuste a la expresión regular especificada en esta sección será ignorada por el *token manager*.
- ***MORE***: A veces es necesario ir construyendo el token que vamos a pasar al analizador sintáctico de forma gradual. Las entradas que se ajustan a la expresión regular son guardadas en un buffer hasta que llega el siguiente *TOKEN* o *SPECIAL_TOKEN*. Después todos los valores almacenados en el buffer son concatenados para formar un solo *TOKEN* o *SPECIAL_TOKEN* y es entonces cuando se envía al analizador sintáctico. En el caso en que una entrada encajara con una expresión regular especificada en la sección *SKIP*, el contenido del *buffer* es desechado.

Tras haber definido el tipo de expresión regular se procede a especificar la expresión propiamente dicha, pero veamos primero algunos ejemplo que ilustren lo visto hasta el momento:

El siguiente programa reconoce comentarios de varias líneas:

```
// Comentarios

SKIP:
{
        "/*":EntreComentarios
}

<EntreComentarios> SKIP:
{
        "*/" : DEFAULT
}

<EntreComentarios> MORE:
{
        <~ [ ] >
}
```

Ejemplo 2.7. Definición de entidades léxicas mediante el empleo de expresiones regulares.

El siguiente ejemplo reconoce cadenas de caracteres y nos devuelve por pantalla su longitud.

```
// Cadena de caracteres.
TOKEN_MGR_DECLS :
{
{int longitudDeCadena;
}

MORE :
{
        "\"" {longitudDeCadena=0;}: EntreComillas
}

<EntreComillas> TOKEN :
{
        <STRLIT:"\"" {System.out.println"Size = " + stringSize);}>: DEFAULT
}

<EntreComillas> MORE :
{
        <~ ["\n","\r"] > {longitudDeCadena++;}
}
```

Ejemplo 2.8. Definición de entidades léxicas mediante el empleo de expresiones regulares.

Como puede verse en los ejemplos 2.7. y 2.8., la combinación del uso de los distintos estados léxicos y de los tipos de expresiones regulares, aporta una gran versatilidad en la definición de las distintas entidades léxicas. En el ejemplo 2.7. se está haciendo uso de dos tipos de entidades léxicas diferentes: SKIP y MORE, y se emplean también dos estados léxicos: <EntreComentarios> y DEFAULT. Por su parte el ejemplo

2.8. hace lo propio, usando dos tipos de entidades: TOKEN y MORE, y otros dos estados léxicos: <EntreComillas> y DEFAULT.

Tras haber aclarado cómo se definen tanto el tipo de entidad léxica como la listas de estados léxicos en que ésta tiene sentido, pasemos a ver realmente cómo se definen haciendo uso de expresiones regulares.

2.5.3.1. EXPRESIONES REGULARES

La especificación de una expresión regular inicia la descripción de las entidades léxicas que forman parte de esta regla de producción. Cada regla de producción puede contener cualquier numero de especificaciones de expresiones regulares.

```
regexpr_espec::=expresion_regular[bloque_java] [“:” identificador_java]
```

Esquema 2.9. Estructura de una especificación de expresión regular.

Cada especificación de expresión regular consta de una expresión regular, seguida por un bloque Java (la acción léxica), el cual es opcional y por el identificador de un estado léxico -el cual también es opcional-. Cuando una entrada se ajusta a una expresión regular, se ejecuta la acción léxica (si es que hay alguna especificada), seguida del código especificado en *void CommonTokenAction (Token t)* (si es que hemos declarado este método). Finalmente si ha sido especificado algún estado léxico, el *token manager* pasa a ese estado.

Hay dos lugares donde se pueden especificar expresiones regulares dentro de la gramática.

- En el interior de la especificación de una expresión regular (una parte de las reglas de producción especificadas mediante expresiones regulares).
- En una unidad de expansión.

```
regular_expression::=java_string_literal
        |  “<” [ [ “#” ] java_identifier “:” ] complex_regular_expresssion_choices “>”
        |  “<” java_identifier “>”
        |  “<” “EOF” “>”
```

Esquema 2.10. Estructura de una expresión regular.

El primer tipo de expresión regular es un literal. La entrada que está siendo analizada encaja con la expresión regular, si el token manager se encuentra en el estado léxico en el que esta expresión regular puede aplicarse y el próximo conjunto de caracteres disponible a la entrada es igual al literal (ignorando posiblemente las mayúsculas).

Una expresión regular puede ser algo más compleja, usando expresiones regulares más complicadas. Podemos colocar una expresión regular entre paréntesis angulares "<...>", opcionalmente podemos etiquetarla con un identificador. Esta etiqueta la podemos utilizar para referenciar a la expresión regular desde unidades de expansión o desde el interior de otras expresiones regulares. Si la expresión regular va precedida de "#", entonces esta expresión regular no puede ser referenciada desde las unidades de expansión, sólo desde el interior de otras expresiones regulares. Cuando la almohadilla ("#") está presente, la expresión regular recibirá el nombre de expresión regular privada.

El propósito de las expresiones regulares privadas es exclusivamente el de facilitar la definición de otras expresiones regulares más complejas.

Consideremos el siguiente ejemplo el cual define literales números en punto flotante en Java.

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

Ejemplo 2.9. Especificación de una expresión regular.

En este ejemplo, el token *FLOATING_POINT_LITERAL* está definido usando otro token llamado EXPONENT. La almohadilla ("#") antes de la etiqueta *EXPONENT* indica que éste existe exclusivamente con el propósito de definir otros *tokens*. La presencia o ausencia de la "#" no afecta en nada a la definición de

FLOATING_POINT_LITERAL, aunque al comportamiento del *token manager* sí. Si la "#" no aparece, el *token manager* reconocerá de forma errónea la cadena E123 como un *token* válido de tipo *EXPONENT* (en lugar de un identificador que es lo que realmente sería).

Podemos apreciar como se emplean listas de caracteres, las cuales se define como una lista de descriptores de caracteres separados por comas y encerrados entre corchetes: ["f","F","d","D"], ["0"-"9"]. Cada descriptor de carácter describe a un único carácter o a un rango de caracteres. Si la lista de caracteres tiene como prefijo el símbolo "~", el conjunto de caracteres representado es cualquier carácter no incluido en el conjunto especificado, así ~["\n"] estará significando cualquier carácter menos el salto de línea.

Un descriptor de carácter puede ser una cadena de un solo carácter("F"), en cuyo caso describe un conjunto con un solo elemento que es el mismo carácter, o son dos cadenas de caracteres con un carácter cada una separados por "-" ("0"-"9"), en cuyo caso describe el conjunto de todos los caracteres entre uno y otro e incluyendo ambos.

2.6. GESTIÓN DE ATRIBUTOS.

Como se ha ido diciendo a lo largo de este capítulo, JavaCC ofrece mecanismos para la gestión de los atributos asociados a las entidades léxicas. En este apartado vamos a hacer una recopilación de todos ellos. Pero en primer lugar debemos hablar de clase token, la cual nos servirá para el posterior desarrollo de cada uno de los mecanismos mencionados.

2.6.1. LA CLASE TOKEN.

La clase Token es el tipo de los objetos token que son creados por el token manager después del escaneo exitoso de la cadena de entrada durante el análisis léxico. Estos objetos son pasados al parser y se accede a las acciones de la gramática JavaCC usualmente retornando el valor del token. Los métodos getToken y getNextToken del token manager, dan acceso a los objetos de este tipo.

Existen dos clases de tokens: **regulares** y **especiales**.

- Los ***tokens regulares*** son los tokens normales que maneja el parser.

- Los ***tokens especiales*** son otros tokens muy usados que aunque no tienen relevancia sintáctica no son descartados, tales como los comentarios.

Cada objeto de tipo token posee los siguientes campos:

- **int kind** Es el índice para esta clase de tokens en el esquema de representación interna de JavaCC. Cuando los tokens tienen etiquetas en el archivo de entrada JavaCC, por ejemplo < MAS: "+" >, dichas etiquetas son utilizadas para generar constantes enteras (int) que pueden usarse en las acciones. El valor 0 siempre se usa para representar el token <EOF> predefinido. Por conveniencia, en el archivo <nombre_archivo>Constants se genera una constante "EOF".

- **int beginLine, beginColumn, endLine, endColumn;** Indican las posiciones de comienzo y final del token, tal como aparecía en el stream de entrada.

- **String image;** Representa la imagen del token como apareció en el stream de entrada.

- **Token next;** Es una referencia al siguiente token regular del stream de entrada. Si es el último token de la entrada, o si el token manager no ha leído tokens después de éste, el campo toma valor null.

  La descripción anterior es válida sólo si el token es un token regular.

- **Token specialToken;** Este campo se utiliza para acceder a los tokens especiales que ocurren antes del token, pero después del token regular (no especial) que lo precede inmediatamente. Si no existen tales tokens especiales, este campo toma el valor null. Cuando hay más de un token especial, este campo se refiere al último de dichos tokens especiales, el cual se refiere al anterior, y así sucesivamente hasta llegar al primer token especial, cuyo campo specialToken es null. Los siguientes campos de tokens especiales se refieren a otros tokens especiales que lo suceden inmediatamente (sin la intervención de un token regular). Si no existe tal token, el campo toma valor null.

- **static final Token newToken(int ofKind);**

```
{
   switch(ofKind){
      default: return new Token();
   }
}
```

Por defecto, su comportamiento consiste en retornar un nuevo objeto Token. Si quieren ejecutarse acciones especiales cuando se construye un token o se crean subclases de la clase Token y se instancian, puede redefinirse este método apropiadamente. La única restricción es que este método retorna un nuevo objeto de tipo Token (o una subclase de Token).

- **public final String toString();**

```
{
   return image;
}
```

Este método retorna el valor de *image*.

Una vez vista la clase Token, vamos a modificar el ejemplo 2.1., añadiéndole una serie de mecanismos de gestión. Se mostrará ahora la gramática de una calculadora que puede hacer uso de variables. Su comportamiento es similar al del ejemplo 2.1. con la salvedad de que ahora vamos a poder declarar variables escribiendo una letra seguido de "=" y de un número. Tras la declaración de una variable, ésta puede ser usada dentro de una expresión. Para ello vamos a crear un estructura en la clase principal sobre la que se van a ir guardando el valor de las distintas variables. Veremos pues cómo trasmitir atributos a lo largo del árbol sintáctico, cómo usar métodos declarados en la clase principal y cómo incrustar código java dentro de las expansiones BNF, en donde se van a poder usar los métodos anteriormente mencionados.

Es importante aclarar que, si bien en el ejemplo 2.1. estábamos construyendo un compilador, ahora se está construyendo un intérprete, pues la salida va a ser una ejecución propiamente dicha.

```
options {
   LOOKAHEAD=1;
}

PARSER_BEGIN(Calculadora )

public class Calculadora {

        public static  Integer[] variables=new Integer[27];

        public static void main(String args[]) throws ParseException {
                Calculadora parser = new Calculadora(System.in);
                while (true) {
                        System.out.print("Introduzca una expresion: ");
                        System.out.flush();
                        try {
                                switch (parser.one_line()) {
                                case -1:
                                        System.exit(0);
                                        default:
                                        break;
                                }
                        } catch (ParseException x) {
                                System.out.println("Finalizando.");
                                throw x;
                         }
                }
        }

        public static void insertaValor(Character indice, Integer valor){
                int i=Character.getNumericValue(indice.charValue()) -
                                                        Character.getNumericValue('A');
                variables[i]=valor;
        }

        public static int valorDeLaVariable(Character var){
                int i=Character.getNumericValue(var.charValue()) -
                                                        Character.getNumericValue('A');
                int resultado=variables[i].intValue();
                return resultado;
        }
}
PARSER_END(Calculadora )

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

TOKEN :
{
        < EOL: "\n" >
}
```

```
TOKEN : /* OPERADORES */
{
        < MAS: "+" >
|       < MENOS: "-" >
|       < POR: "*" >
|       < ENTRE: "/" >
|       < IGUAL: "=" >
}

TOKEN :
{
        < CONSTANTE: ( <DIGITO> )+ >
|       < #DIGITO: ["0" - "9"] >
}
TOKEN :
{
        <VARIABLE: <LETRA>>
|       <#LETRA:["A"-"Z"]>
}

int one_line() :
{int resultado;}
{
  LOOKAHEAD(2)
        resultado=suma(){System.out.println("El resultado es: "+resultado);}
        <EOL>{ return 1; }
  | decl_var() <EOL>{return 0;}
  |<EOL>{ return 0; }
  |<EOF>
{ return -1; }
}
int suma() :
{int resultado;
 int aux; }
{
  resultado=termino()
  (  <MAS> aux=termino(){resultado=resultado+aux;}
   |<MENOS> aux=termino(){resultado=resultado-aux;}
  )*
  {return resultado;}
}

int termino() :
{ int resultado;
  int aux;}
{
  resultado=unario()
  (  <POR> aux=unario(){resultado=resultado*aux;}
   |<ENTRE> aux=unario(){resultado=resultado/aux;}
  )*
  {return resultado;}
}
```

```
int unario() :
{ int resultado;}
{
  (         <MENOS> resultado=elemento(){resultado=0-resultado;}
   |        resultado=elemento()
  )
            {return resultado;}
}

int elemento() :
{Integer valor;
 int resultado;
 Character aux;}
{
  (   <CONSTANTE>       {valor=new Integer(token.image);
                         resultado=valor.intValue();
                         }
   | <VARIABLE>         {aux=new Character(token.image.charAt(0));
                         resultado=valorDeLaVariable(aux);
                         }
   | "(" resultado=suma() ")"
  )
            {return resultado;}
}
void decl_var() :
{Integer valor;
 Character indice;}
{
            <VARIABLE>{indice=new Character(token.image.charAt(0));}
            <IGUAL>
            <CONSTANTE>{valor=new Integer(token.image);}
            {insertaValor(indice,valor);}
}
```

Ejemplo 2.10. Calculadora modificada con gestión de atributos.

Antes de comenzar a analizar los distintos mecanismos de gestión de atributos de que disponemos, vamos a comentar un aspecto importante mostrado en el ejemplo 2.10. En la definición del No Terminal one_line() se ha hecho uso de la opción LOOKAHEAD (2). Como se comentó anteriormente, JavaCC permite que el parser se vuelva LL(k) en algunas zonas en concreto, manteniendo el resto en LL(1). Eso es justo lo que se muestra aquí. Mediante el empleo de la opción LOOKAHEAD, se nos permite modificar el tipo de parser, tanto a nivel general como a nivel local.

Este aspecto es muy importante y es una de las características más destacables de JavaCC.

2.6.2. MECANISMOS PARA LA GESTIÓN DE ATRIBUTOS.

-Añadiendo acciones léxicas en la definición de un token mediante bloques de código java. Como puede verse en el ejemplo 2.8. en donde se define un token de la siguiente manera:

```
<EntreComillas> TOKEN :
{
        <STRLIT:"\"" {System.out.println"Size = " + stringSize);}>: DEFAULT
}
```

Como podemos ver, tras el reconocimiento de la entidad léxica se realiza una acción, que en este caso ha sido la de mostrar una línea por pantalla, pero del mismo modo se hubiera podido recoger el contenido de token.image y guardarlo en alguna variable del programa o cualquier otra clase de acción que conlleve el reconocimiento del atributo asociado. Cuando una entrada se ajusta a una expresión regular, se ejecuta la acción léxica (si es que hay alguna especificada).

- **Declarando el método *void CommonTokenAction (Token t)***. Éste se va a ejecutar siempre después de que una entrada se ajuste a alguna de las expresiones regulares usadas para definir las distintas entidades léxicas. De este modo podríamos hacer algo como:

```
void CommonTokenAction (Token t){
                numeroDePalabras++;
                atributoDisponible=t.image;
}
```

Ejemplo 2.11. Uso de CommonTokenAction.

En el ejemplo 2.11., númeroDePalabras y atributoDisponible son variables declaradas en la clase principal sobre las cuales trabajamos. Las posibilidades que se nos ofrecen mediante el empleo de este método son enormes, pudiendo hacer cualquier clase de comprobación sobre la cadena de entrada y en base a ella trabajar de una forma u otra.

-Haciendo uso de los parámetros de entrada y tipos de retorno de los No Terminales. Como hemos visto, los No Terminales se van a declarar como métodos en java y por lo tanto tienen sus correspondientes parámetros de entrada y de salida. Éstos pueden ser usados para transmitir hacia arriba o hacia abajo en el árbol sintáctico atributos asociados a entidades léxicas. En el ejemplo 2.10. podemos ver el empleo de este mecanismo:

```
int unario() :
{ int resultado;}
{
  (       <MENOS> resultado=elemento(){resultado=0-resultado;}
  |       resultado=elemento()
  )
          {return resultado;}
}
```

Ejemplo 2.12. Uso de los parámetros de los No Terminales.

Vemos en este ejemplo cómo tras el reconocimiento del No terminal nos es devuelto un entero, generado dentro del método. Éste es recogido por una variable declarada en la primera zona de la parte derecha del No Terminal unario(). Los valores devueltos sólo podrán ser recogidos por variable declaradas previamente en esta zona, para realizar cualquier otro tipo de operaciones sobre estas variables se debe hacer bloques de código java incrustados en la expansión BNF como se verá a continuación.

Una vez declarado un tipo de retorno en el No Terminal, éste debe ser devuelto tras el reconocimiento del mismo. Para ello existen dos posibilidades: o bien incrustar un bloque de código java al final de la expansión BNF que devuelva una variable del tipo definido como tipo de retorno del No Terminal, o bien hacerlo tras cada una de las posibilidades de la expansión. En definitiva, se debe asegurar que siempre se devuelva lo esperado. En el ejemplo 2.12. se ha optado por hacerlo al final: return resultado, donde resultado es una variable declarada previamente del mismo tipo que el de retorno de unario().

Mediante el empleo de este mecanismo se están transmitiendo los diferentes atributos a lo largo del árbol sintáctico. Dentro de un cada uno de los métodos vamos a poder trabajar con ellos como se verá a continuación.

-Incrustando bloques de código java dentro de las expansiones BNF. Como ya se ha dicho, en el interior de una expansión BNF se pueden incluir bloques de código java

que realicen aquello que se desee. Dentro de estos bloques se puede hacer uso tanto de las variables y métodos definidos dentro del No Terminal, en la zona dedicada a tal fin, como de métodos propios de la clase principal, declarados dentro de la zona delimitada por PARSER_BEGIN ..... PARSER_END. Esta posibilidad asociada al uso de la clase token es el principal mecanismo de gestión de atributos. De hecho la mayoría de acciones semánticas se van a realizar de esta forma. Las posibilidades que se nos brindan ahora son enormes. En el ejemplo 2.10. gran parte de la gestión de atributos se ha desarrollado de esta forma. Se combinan el uso de métodos declarados en la clase Calculadora, insertaValor y valorDeLaVariable, con empleo de token.image. Veamos un trozo en concreto:

```
int elemento() :
{Integer valor;
 int resultado;
 Character aux;}
{
  (   <CONSTANTE>       {valor=new Integer(token.image);
                         resultado=valor.intValue();
                        }
   | <VARIABLE>         {aux=new Character(token.image.charAt(0));
                         resultado=valorDeLaVariable(aux);
                        }
   | "(" resultado=suma() ")"
  )
          {return resultado;}
}
```

Ejemplo 2.13. Empleo de bloques de código Java en las expansiones BNF

Podemos apreciar cómo, tras el reconocimiento de una entrada, se han insertado acciones que conllevan una interpretación de la misma como atributos asociados a las diversas entidades léxicas. Es en esta parte donde recae casi toda la acción semántica que el parser lleva asociado. Se puede observar cómo el empleo de la clase token se hace indispensable para el desarrollo de esta tarea.

En el siguiente capítulo se va a mostrar mediante diagramas Conway la gramática que se ha definido para nuestro lenguaje (las Especificaciones Algebraicas). Y se va a mostrar las estructuras que se generarían en memoria y cómo se van a usar para reducir un término.