

Procesadores del Lenguaje – Trabajo Final

Desarrollo de un traductor de gramáticas en forma de Backus-Naur a
forma normal de Chomsky

Memoria Técnica



Universidad de Huelva



Realizado por:

Álvaro Esteban Muñoz

Índice de contenido

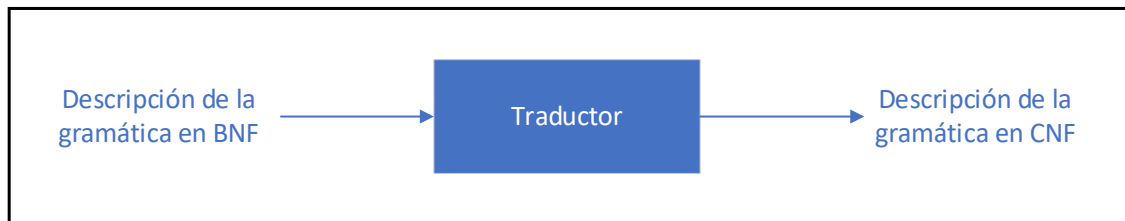
Introducción	4
Análisis Léxico.....	4
Análisis Sintáctico.....	6
Análisis Semántico.....	7
Árbol de Sintaxis Abstracta propuesto.....	7
Transformación de la gramática en una gramática atribuida	10
Gestión de errores.....	11
Algoritmo de traducción de la gramática.....	13
Primer paso de la traducción	14
Segundo paso de la traducción	15
Tercer paso de la traducción.....	16
Cuarto paso de la traducción	17
Funciones auxiliares empleadas en el algoritmo de traducción	17
Pruebas sobre la aplicación.....	20

Índice de ilustraciones

<i>Ilustración 1: Implementación Léxica (1)</i>	5
<i>Ilustración 2: Implementación Léxica (2)</i>	5
<i>Ilustración 3: Implementación Sintáctica</i>	6
<i>Ilustración 4: Implementación de Gramatica.java</i>	7
<i>Ilustración 5: Implementación de Definición.java</i>	8
<i>Ilustración 6: Implementación de Regla.java</i>	8
<i>Ilustración 7: Implementación de Symbol.java</i>	9
<i>Ilustración 8: Implementación de TSymbol.java</i>	9
<i>Ilustración 9: Implementación de NTSymbol.java</i>	9
<i>Ilustración 10: Implementación Semántica (1)</i>	10
<i>Ilustración 11: Implementación Semántica (2)</i>	10
<i>Ilustración 12: Método skipTo para la gestión de errores</i>	11
<i>Ilustración 13: Ejemplo de tokens de sincronización</i>	11
<i>Ilustración 14: Método para generar la traducción</i>	13
<i>Ilustración 15: Primera parte del primer paso de traducción</i>	14
<i>Ilustración 16: Segunda parte del primer paso de traducción</i>	14
<i>Ilustración 17: Segundo paso de traducción</i>	15
<i>Ilustración 18: Tercer paso de traducción</i>	16
<i>Ilustración 19: Cuarto paso de traducción</i>	17
<i>Ilustración 20: Método borraSimb()</i>	17
<i>Ilustración 21: Método sustituyeSimb() - 1</i>	18
<i>Ilustración 22: Método sustituyeSimb() - 2</i>	18
<i>Ilustración 23: Método creaSimbolo()</i>	18
<i>Ilustración 24: Método getNumTerminales()</i>	19
<i>Ilustración 25: Ejemplo fichero de entrada</i>	20
<i>Ilustración 26: Salida del traductor (Explorador de archivos)</i>	20
<i>Ilustración 27: Salida del traductor</i>	21
<i>Ilustración 28: Ejemplo de error en la entrada del traductor</i>	21
<i>Ilustración 29: Salida de errores del traductor (Explorador de archivos)</i>	22
<i>Ilustración 30: Salida de errores del traductor</i>	22

Introducción

El objetivo de nuestra aplicación es la generación de una gramática descrita en Forma Normal de Chomsky (CNF o Chomsky Normal Form) a partir de su descripción en forma de Backus-Naur (BNF o Backus-Naur Form). En otras palabras, estamos ante un traductor que toma como entrada la descripción de una gramática en BNF y la traduce a su descripción en CNF.



El traductor se va a implementar usando el lenguaje de programación Java y haciendo uso de un metacompilador llamado JavaCC. El metacompilador nos ayudará a resolver el análisis léxico, el análisis sintáctico y nos ayudará con gran parte del análisis semántico.

Análisis Léxico

El metacompilador JavaCC hace muy sencilla la implementación del análisis léxico ya que solo tendremos que plasmar las expresiones regulares usadas para describir cada token en un fichero “.jj” y JavaCC se encargará de crear todas las clases relacionadas.

Especificación	Expresión regular
blanco	(" " "\r" "\n" "\t")
comentario	"/*" (("*")* ~["*", "/" "/"])* ("*")+ "/"
NOTERMINAL	["_", "a"- "z", "A"- "z"] (["_", "a"- "z", "A"- "Z", "0"- "9"])*
TERMIINAL	"<" ["_", "a"- "z", "A"- "z"] (["_", "a"- "z", "A"- "Z", "0"- "9"])* ">"
EQ	"::="
BAR	" "
SEMICOLON	";"

La parte en del fichero “BNF2CNF_Parser.jj” en la que se describe la especificación léxica se divide dependiendo de la acción a tomar por el AFN generado por el metacompilador, usando la cláusula “SKIP” el AFN omitirá la cadena y con la cláusula TOKEN el AFN generará un token.

- Cadenas para omitir:

```
/* ESPECIFICACIÓN LÉXICA DE LA GRAMÁTICA */

/* Blanco */

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

/* Comentario */

SKIP:
{
    < COMENTARIO: "/"* ( ("*")* ~["*", "/" ] | "/" )* ("*")+ "/" >
}
```

Ilustración 1: Implementación Léxica (1)

- Generadores de token:

```
/* Identificadores */

TOKEN:
{
    < NOTERMINAL: ["_", "a"-"z", "A"-"Z"] ( ["_", "a"-"z", "A"-"Z", "0"-"9"] )* >
    |
    < TERMINAL: "<" ["_", "a"-"z", "A"-"Z"] ( ["_", "a"-"z", "A"-"Z", "0"-"9"] )* ">" >
}

/* Operadores */

TOKEN:
{
    < EQ: "[:=" >
}

/* Separadores */

TOKEN:
{
    < BAR: "|" >
    |
    < SEMICOLON: ";" >
}
```

Ilustración 2: Implementación Léxica (2)

Análisis Sintáctico

La implementación del analizador sintáctico es tan sencilla como la del analizador léxico, de hecho, se realiza dentro del mismo fichero “.jj” que el analizador léxico. La parte del fichero “BNF2CNF_Parser.jj” dedicada al analizador sintáctico está colocada justo a continuación del analizador léxico para que sea fácil de leer.

La especificación sintáctica de nuestro lenguaje a ser reconocido es la siguiente:

- Gramatica ::= (Definicion)*
- Definicion ::= NOTERMINAL EQ ListaReglas SEMICOLON
- ListaReglas ::= Regla (BAR Regla)*
- Regla ::= (NOTERMINAL | TERMINAL)*

```
/* ESPECIFICACIÓN SINTÁCTICA DE LA GRAMÁTICA */

void Gramatica() :
{
{
( Definicion() )*
}
}

void Definicion() :
{
{
<NOTERMINAL> <EQ> ListaReglas() <SEMICOLON>
}
}

void ListaReglas() :
{
{
Regla() ( <BAR> Regla() )*
}
}

void Regla() :
{
{
( <NOTERMINAL> | <TERMINAL> )*
}
}
```

Ilustración 3: Implementación Sintáctica

Análisis Semántico

Para la implementación del análisis semántico se complican un poco las cosas, puesto que debemos pensar en una forma de estructurar la información que vamos a almacenar en la memoria, aquella información sobre el lenguaje procesado que luego usaremos para generar el código de salida.

Árbol de Sintaxis Abstracta propuesto

La información que mantendremos en memoria se estructurará según las siguientes clases:

- **Gramática:** Contendrá una estructura *ArrayList* de objetos de la clase *Definición*
- **Definición:** Constatará de dos partes, un objeto de la clase *NTSymbol* y una estructura *ArrayList* de objetos de la clase *Regla*.
- **Regla:** Contendrá una estructura *ArrayList* de objetos de tipo *Symbol*.
- **Symbol:** Interfaz que implementarán tanto los *NTSymbol* como los *TSymbol*, de esta forma podremos abarcarlos en el mismo *ArrayList* independientemente del tipo de su instancia.
- **NTSymbol:** Contiene una *String* que guardará el nombre del símbolo, en otras palabras, el lexema usado para representar a dicho símbolo no terminal.
- **TSymbol:** Es exactamente igual que un *NTSymbol*, creamos una clase diferente para poder diferenciar el tipo de símbolo que estamos leyendo.

**Todas las clases tienen un método toString() que hemos implementado para poder depurar el programa de forma más intuitiva.*

```
1 package bnf2cnf.ast;
2
3 import java.util.ArrayList;
4
5
6 public class Gramatica {
7
8     private ArrayList<Definicion> Definiciones;
9
10    public Gramatica(ArrayList<Definicion> Definiciones) {
11        this.Definiciones = Definiciones;
12    }
13
14    @Override
15    public String toString() {
16
17        String G;
18
19        G = "Gramática: \n ";
20
21        for (Definicion definicion : Definiciones) {
22            G = G + definicion.toString();
23        }
24
25        return G;
26    }
27 }
28
29
```

Ilustración 4: Implementación de Gramatica.java

```

1 package bnf2cnf.ast;
2
3 import java.util.ArrayList;
4
5
6
7 public class Definicion {
8
9     private NTSymbol NT;
10    private ArrayList<Regla> ListaReglas;
11
12    public Definicion(NTSymbol NT, ArrayList<Regla> ListaReglas) {
13        this.NT = NT;
14        this.ListaReglas = ListaReglas;
15    }
16
17    @Override
18    public String toString() {
19
20        String D = NT.toString() + " ::= ";
21
22        for (Regla regla : ListaReglas) {
23            D = D + regla.toString() + "\n | ";
24        }
25        D = D + "\n ; \n";
26
27        return D;
28    }
29 }
30
31

```

Ilustración 5: Implementación de Definición.java

```

1 package bnf2cnf.ast;
2
3 import java.util.ArrayList;
4
5
6
7 public class Regla {
8
9     private ArrayList<Symbol> Simbolos;
10
11    public Regla(ArrayList<Symbol> Simbolos) {
12        this.Simbolos = Simbolos;
13    }
14
15    @Override
16    public String toString() {
17
18        String R = "";
19
20        for (Symbol symbol : Simbolos) {
21            R = R + symbol.toString() + " ";
22        }
23
24        return R;
25    }
26 }
27
28

```

Ilustración 6: Implementación de Regla.java


```

1 package bnf2cnf.ast.Symbols;
2
3 public interface Symbol {
4
5     @Override
6     public String toString();
7 }
8

```

Ilustración 7: Implementación de Symbol.java

```

1 package bnf2cnf.ast.Symbols;
2
3 public class TSymbol implements Symbol {
4
5     private String name;
6
7     public TSymbol(String name) {
8         this.name = name;
9     }
10
11     @Override
12     public String toString() {
13         return name;
14     }
15 }
16

```

Ilustración 8: Implementación de TSymbol.java

```

1 package bnf2cnf.ast.Symbols;
2
3 public class NTSymbol implements Symbol {
4
5     private String name;
6
7     public NTSymbol(String name) {
8         this.name = name;
9     }
10
11     @Override
12     public String toString() {
13         return name;
14     }
15 }
16
17

```

Ilustración 9: Implementación de NTSymbol.java

Transformación de la gramática en una gramática atribuida

Una vez tenemos creada la estructura que tendrá el árbol de sintaxis abstracta, tendremos que convertir nuestra gramática en una gramática atribuida, de forma que pueda almacenar la información en las clases que hemos implementado.

Para transformar nuestra gramática en una gramática atribuida tendremos que desarrollar el **ETDS** (esquema de traducción dirigido por la sintaxis) en nuestro fichero de javaCC.

```
67 Gramatica Gramatica() :
68 {
69     Gramatica G;
70     Definicion D = null;
71     ArrayList<Definicion> Definiciones = new ArrayList<Definicion>();
72 }
73 {
74     ( D = Definicion() { Definiciones.add(D); }
75     )*
76     {
77         G = new Gramatica(Definiciones);
78         return G;
79     }
80 }
81
82 Definicion Definicion() :
83 {
84     Definicion D;
85     Token tk;
86     NTSymbol NT;
87     ArrayList<Regla> LR = new ArrayList<Regla>();
88 }
89 {
90     tk = <NOTERMINAL> { NT = new NTSymbol(tk.image); } <EQ> ListaReglas(LR) <SEMICOLON>
91     {
92         D = new Definicion(NT, LR);
93         return D;
94     }
95 }
96
97 void ListaReglas(ArrayList<Regla> LR) :
98 {
99     Regla R = null;
100 }
101 {
102     R = Regla(LR)
103     (
104         <BAR> R = Regla(LR)
105     )*
106 }
```

Ilustración 10: Implementación Semántica (1)

```
108 Regla Regla(ArrayList<Regla> LR) :
109 {
110     Token tk = null;
111     Symbol S;
112     Regla R;
113     ArrayList<Symbol> Simbolos = new ArrayList<Symbol>();
114 }
115 {
116     (
117         tk = <NOTERMINAL> { S = new NTSymbol(tk.image); Simbolos.add(S); }
118         | tk = <TERMINAL> { S = new TSymbol(tk.image); Simbolos.add(S); }
119     )*
120     {
121         R = new Regla(Simbolos);
122         LR.add(R);
123         return R;
124     }
125 }
```

Ilustración 11: Implementación Semántica (2)

Gestión de errores

En JavaCC la gestión de errores léxicos se hace automáticamente así que no tendremos que preocuparnos por esa parte, simplemente definiremos la forma de recuperarnos de errores sintácticos y semánticos.

Para recuperarnos de los errores hemos creado la función skipTo() que nos permite saltarnos los tokens de entrada hasta llegar a un token de sincronismo.

```
JAVACODE
void skipTo(int[] left, int[] right)
{
    Token prev = getToken(0);
    Token next = getToken(1);
    boolean flag = false;
    if(prev.kind == EOF || next.kind == EOF) flag = true;
    for(int i=0; i<left.length; i++) if(prev.kind == left[i]) flag = true;
    for(int i=0; i<right.length; i++) if(next.kind == right[i]) flag = true;
    while(!flag)
    {
        getNextToken();
        prev = getToken(0);
        next = getToken(1);
        if(prev.kind == EOF || next.kind == EOF) flag = true;
        for(int i=0; i<left.length; i++) if(prev.kind == left[i]) flag = true;
        for(int i=0; i<right.length; i++) if(next.kind == right[i]) flag = true;
    }
}
```

Ilustración 12: Método skipTo para la gestión de errores

Una vez creado este método, lo usaremos para recuperarnos de los errores, tendremos que definir los tokens de sincronismo a izquierda y derecha al comienzo de cada clase.

```
Gramatica Gramatica() :
{
    int[] lsync = { NOTERMINAL, TERMINAL, SEMICOLON, BAR, EQ };
    int[] rsync = { EOF };
    Gramatica G;
    Definicion D = null;
    ArrayList<Definicion> Definiciones = new ArrayList<Definicion>();
}
```

Ilustración 13: Ejemplo de tokens de sincronización

Los tokens de sincronismo a izquierda (lsync) y derecha (rsync) quedarán de la siguiente forma:

- Gramática:

Izquierda → <NOTERMINAL>, <TERMINAL>, <SEMICOLON>, <BAR>, <EQ>

Derecha → <EOF>

- Definición:

Izquierda → <SEMICOLON>

Derecha → []

- ListaReglas:

Izquierda → []

Derecha → <SEMICOLON>

- Regla:

Izquierda → []

Derecha → <BAR>, <SEMICOLON>

Algoritmo de traducción de la gramática

Para la implementación del algoritmo de traducción hemos creado una clase nueva “*ChomskyGenerato.java*” que se encargará de recibir el árbol de sintaxis abstracta de la gramática analizada y la traducirá al nuevo árbol de sintaxis abstracta de la gramática en forma normal de Chomsky. El algoritmo de traducción es algo largo, pero se puede dividir en cuatro pasos que hemos implementado de la siguiente forma:

```
public abstract class ChomskyGenerator {  
  
    private static Gramatica g_cnf; // Gramática en forma normal de Chomsky  
  
    public static Gramatica generarCNF(Gramatica G) {  
  
        // Debemos hacer una copia de la gramática, no modificarla  
        g_cnf = G.clone();  
  
        ChomskyGenerator.traducePrimerPaso(); // Primer paso del algoritmo  
        ChomskyGenerator.traduceSegundoPaso(); // Segundo paso del algoritmo  
        ChomskyGenerator.traduceTercerPaso(); // Tercer paso del algoritmo  
        ChomskyGenerator.traduceCuartoPaso(); // Cuarto paso del algoritmo  
  
        return g_cnf;  
    }  
}
```

Ilustración 14: Método para generar la traducción

Primer paso de la traducción

El primer paso consiste en eliminar las reglas lambda, hemos dividido este trabajo en dos partes, primero buscamos aquellas definiciones que contengan alguna regla lambda y almacenamos el símbolo NT de la definición en un **ArrayList**. Tras esto, recorreremos el **ArrayList** de los símbolos de las definiciones que han sido modificadas y si encontramos una regla que derive en dicho símbolo, crearemos una copia de esa regla, pero eliminando el símbolo que se encuentra en el array de la regla copiada, así tendremos una regla que deriva en el símbolo y otra que no.

```
/**
 * PRIMER PASO: ELIMINAR LAS REGLAS LAMBDA
 * Eliminar las reglas A → λ y para cada regla en la que aparezca el símbolo A se generan dos reglas, una con A y otra sin A.
 */
private static void traducePrimerPaso() {
    // Extraemos las definiciones de la gramática
    ArrayList<Definicion> defs_cnf = g_cnf.getDefiniciones();

    // Guardaremos los NTSymbols que deberemos tener en cuenta para añadir las nuevas definiciones
    ArrayList<NTSymbol> simbolosModificados = new ArrayList<NTSymbol>();

    // Para cada definición de la gramática, buscamos si tiene regla lambda
    for (Definicion d : defs_cnf) {
        ArrayList<Regla> reglas = d.getListaReglas(); // Extraemos su lista de reglas
        int i = 0; // Índice del búsqueda de la regla lambda en la definición actual
        boolean encontrada = false; // Flag --> ¿Hemos encontrado la regla lambda de esta definición?

        // Buscamos si la definición contiene una regla lambda en su lista de reglas
        while (i < reglas.size() && !encontrada) {
            Regla reglaActual = reglas.get(i); // Obtenemos la regla actual

            if (reglaActual.getSimbolos().isEmpty()) { // Si la regla tiene un array de símbolos vacíos == Es una regla lambda
                // Guardaremos el NTSymbol de la regla en un arrayList para cambiar aquellas reglas que lo contengan más adelante.
                simbolosModificados.add(d.getSimbolo());

                reglas.remove(i); // Borramos la regla lambda
                encontrada = true; // Regla encontrada, no seguimos buscando
            }
            i++;
        }
    }
}
```

Ilustración 15: Primera parte del primer paso de traducción

```
// Para cada definición miramos si hemos eliminado una regla lambda
for (Definicion d : defs_cnf) {
    ArrayList<Regla> reglasActual = d.getListaReglas(); // Extraemos las reglas de la definición actual

    // Para cada regla de la definición...
    for (int i=0; i<reglasActual.size(); i++) {
        ArrayList<Symbol> simbolosActual = reglasActual.get(i).getSimbolos(); // Extraemos los símbolos de la regla actual

        // Para cada símbolo de la regla...
        for (Symbol sActual : simbolosActual) {
            // Si nuestro sActual coincide con alguno del array de símbolos modificados, crearemos una nueva regla igual que
            // la regla actual pero eliminando el sActual.
            for (NTSymbol sModifActual : simbolosModificados) {
                // Comparación símbolo actual == símbolo modificado actual
                if (sActual.equals(sModifActual)) {
                    Regla regModif = reglasActual.get(i).clone(); // Clonamos la regla actual
                    regModif.borraSimb(sActual); // Pero borramos el símbolo actual
                    reglasActual.add(regModif); // Lo añadimos a la lista de reglas de la definición actual
                }
            }
        }
    }
}
```

Ilustración 16: Segunda parte del primer paso de traducción

Segundo paso de la traducción

En el segundo paso debíamos aislar los símbolos terminales a definiciones que deriven directamente en ellos, para lograr esto, buscábamos aquellas reglas dentro de cada definición que contuvieran símbolos terminales y seguidamente los sustituíamos por un símbolo no terminal nuevo o por uno ya existente si fuera posible.

```
private static void traduceSegundoPaso() {  
    NTSymbol NTDefAux;  
  
    // Array de símbolos auxiliares de los que crearemos reglas aisladas  
    ArrayList<TSymbol> auxSymbols2 = new ArrayList<TSymbol>();  
    ArrayList<NTSymbol> auxDerivs = new ArrayList<NTSymbol>();  
  
    ArrayList<Definicion> defs_cnf = g_cnf.getDefiniciones(); // Extraemos la definiciones de la gramática  
  
    // Recorreremos todas las definiciones buscando aquellas reglas que contengan símbolos terminales aislados,  
    for (int i=0; i<defs_cnf.size(); i++) { // Para cada definición...  
  
        ArrayList<Regla> reglasActual = defs_cnf.get(i).getListaReglas(); // Extraemos la lista de reglas de la definición actual  
  
        for (int j=0; j<reglasActual.size(); j++) { // Para cada regla...  
  
            Regla reglaActual = reglasActual.get(j); // Extraemos la regla actual  
            ArrayList<Symbol> simbolosActual = reglaActual.getSimbolos(); // Extraemos el array de símbolos de la regla actual  
            int contador = reglaActual.getNumTerminales(); // Contamos los símbolos terminales en la lista de símbolos  
  
            // Si la regla contiene un símbolo terminal --> (contador > 0 && simbolosActual.size() > 1)  
            if (contador > 0 && simbolosActual.size() > 1) {  
  
                int k = 0; // Índice del bucle  
                int encontrado = 0; // Número de símbolos terminales encontrados  
  
                // Mientras no hayamos sustituido todos los símbolos terminales de la regla, iteramos  
                while (k<simbolosActual.size() && encontrado<contador) {  
  
                    // Si el símbolo es terminal, lo hemos encontrado  
                    if (simbolosActual.get(k) instanceof TSymbol) {  
                        encontrado++;  
                        reglaActual.sustituyeSimb(g_cnf, (TSymbol) simbolosActual.get(k));  
                    }  
  
                    k++;  
                }  
            }  
        }  
    }  
}
```

Ilustración 17: Segundo paso de traducción

Tercer paso de la traducción

En el tercer paso de la traducción debíamos expandir las reglas que tuvieran un único símbolo no terminal en la parte derecha. Conseguir esto se resume en buscar en cada definición aquellas reglas con un único símbolo terminal a la derecha y añadir las reglas en las que deriva ese símbolo terminal a la definición que estamos comprobando en ese momento.

```
/**
 * TERCER PASO: EXPANDIR LAS REGLAS CON UN ÚNICO SÍMBOLOS
 * Sustituir las reglas de tipo A → B (es decir, las reglas con un único símbolo no terminal en la parte derecha). Para ello,
 * por cada regla B → a se crea una nueva regla A → a y por cada regla B → C D E se crea una regla A → C D E.
 */
private static void traduceTercerPaso() {

    ArrayList<Definicion> defs_cnf = g_cnf.getDefiniciones(); // Extraemos las definiciones de la gramática
    boolean reglasActualizadas = true; // Flag --> ¿Hemos actualizado las reglas en la última vuelta del bucle?

    while (reglasActualizadas) {

        reglasActualizadas = false; // Marcamos las reglas como no actualizadas

        // Recorremos la gramática buscando aquellas reglas que tengan un único símbolo que además sea no terminal
        for (Definicion definicion : defs_cnf) {

            ArrayList<Regla> reglasActual = definicion.getListaReglas();

            for (int i=0; i<reglasActual.size(); i++) { // Para cada regla de la definición...

                Regla reglaActual = reglasActual.get(i);

                if (reglaActual.getNumTerminales() == 0 && reglaActual.getSimbolos().size() == 1) {
                    // Estamos ante una regla con un único símbolo que además es no terminal, tendremos que derivar la regla
                    // hasta alcanzar un símbolo terminal

                    // Obtenemos la derivación de la regla
                    ArrayList<Regla> reglasDerivadas = g_cnf.derivaRegla(reglaActual.getSimbolos().get(0));

                    reglasActual.remove(i); // Borramos la regla actual

                    // Añadimos las reglas una vez derivadas
                    for (int j=0; j<reglasDerivadas.size(); j++) reglasActual.add(reglasDerivadas.get(j));

                    reglasActualizadas = true; // Marcamos que hemos actualizado las reglas
                }
            }
        }
    }
}
```

Ilustración 18: Tercer paso de traducción

Cuarto paso de la traducción

Para finalizar, teníamos que trocear las reglas que contuvieran más de dos símbolos a la derecha, así que recorríamos las definiciones buscando aquellas reglas con más de dos símbolos y si este era el caso, almacenábamos los símbolos sobrantes en un Array que eran sustituidos por un símbolo nuevo.

```
/**
 * CUARTO PASO: TROCEAR LAS REGLAS CON MÁS DE DOS SÍMBOLOS
 * Por cada regla con más de dos símbolos no terminales a la derecha, A → B C D ..., se crean nuevos símbolos terminales N1, N2, ...
 * y se sustituye la regla por A → B N1, N1 → C N2, N2 → D...
 */
private static void traduceCuartoPaso() {
    ArrayList<Definicion> defs_cnf = g_cnf.getDefiniciones(); // Extraemos las definiciones de la gramática

    // Buscamos todas las definiciones que contengan reglas con más de dos símbolos
    for (int j=0; j<defs_cnf.size(); j++) { // Para cada definición de la gramática...

        ArrayList<Regla> reglasActual = defs_cnf.get(j).getListaReglas(); // Extraemos el array de reglas de la definición actual

        for (Regla regla : reglasActual) { // Para cada regla de la definición actual...

            //System.out.println("Una regla troceada después...\n" + g_cnf.toString());
            if (regla.getSimbolos().size() > 2) {

                // La regla tiene más de dos símbolos y debe ser troceada.
                NTSymbol simboloNuevaDef = Regla.creaSimbolo(); // Creamos el símbolo NT que derivará en la lista de símbolo
                ArrayList<Symbol> simbolosSobrantes = new ArrayList<Symbol>(); // Creamos el array de símbolos que deberán ser apartados

                // Pasamos cada símbolo con índice mayor o igual que 1 (1, 2, 3, 4...) a un array aparte.
                // Añadimos el símbolo al array de símbolos sobrantes
                // Como estamos modificando el tamaño del array, tendremos que guardar el tamaño inicial
                int numSimbolos = regla.getSimbolos().size();

                for (int i=1; i<numSimbolos; i++) {
                    simbolosSobrantes.add(regla.getSimbolos().get(i));
                    regla.getSimbolos().remove(i);
                }

                // Sustituimos los símbolos a partir de la segunda posición por el símbolo nuevo
                regla.getSimbolos().add(1, simboloNuevaDef);

                Regla reglaSimbolosSobrantes = new Regla(simbolosSobrantes); // Creamos la regla de la definición troceada
                ArrayList<Regla> reglasDefinicion = new ArrayList<Regla>(); // Creamos el array de reglas de la definición troceada
                reglasDefinicion.add(reglaSimbolosSobrantes); // Añadimos la regla en el array

                Definicion defTroceada = new Definicion(simboloNuevaDef, reglasDefinicion); // Creamos la definición que será añadida a la
                defs_cnf.add(defTroceada);
            }
        }
    }
}
```

Ilustración 19: Cuarto paso de traducción

Funciones auxiliares empleadas en el algoritmo de traducción

Para realizar ciertas funciones y no perdernos en la complejidad del código, hemos implementado unas funciones sobre las que hemos delegado algunas tareas importantes o repetitivas, a continuación, se puede observar una lista de dichas funciones y su implementación.

```
/**
 * Borra de la lista de símbolos que componen la regla el símbolo pasado por parámetro
 * @param sActual Símbolo a ser eliminado de la lista de símbolos de la regla
 */
public void borraSimb(Symbol sActual) {
    for (int i=0; i<simbolos.size(); i++) {
        if (simbolos.get(i).equals(sActual)) simbolos.remove(i);
    }
}
```

Ilustración 20: Método borraSimb()

```

/**
 * Sustituye el símbolo terminal suministrado por parámetro por un símbolo no terminal
 * @param G
 * @param sustituido símbolo terminal que será sustituido en la regla
 * @return el símbolo no terminal que sustituye al suministrado por parámetro
 */
public void sustituyeSimb(Gramatica G, TSymbol sustituido) {

    NTSymbol sustituto = null;           // Símbolo no terminal que será encargado de sustituir al símbolo terminal suministrado
    boolean creadoNuevo = false;        // Flag --> ¿Se ha creado una nueva definición?

    ArrayList<Definicion> defs = G.getDefiniciones();

    // Comprobamos que no exista ya una definición que derive de forma aislada en sustituido
    Definicion derivacion = G.existeDerivacionAislada(sustituido);
    if (derivacion != null) {
        // Existe una definición que deriva en sustituido de forma aislada, usaremos su NT como sustituto
        sustituto = derivacion.getSimbolo();
    } else {
        // No existe una definición que deriva en sustituido de forma aislada, creamos un símbolo no terminal nuevo
        sustituto = Regla.creaSimbolo();

        // Creamos la definición aislada nueva
        ArrayList<Symbol> simbolosDerivacion = new ArrayList<Symbol>(); // Símbolos de la derivación aislada
        simbolosDerivacion.add(sustituto); // Añadimos el sustituto

        Regla reglaDerivacion = new Regla(simbolosDerivacion); // Creamos la regla a partir del array de símbolos
        ArrayList<Regla> reglasDerivacion = new ArrayList<Regla>(); // Creamos el array de reglas
        reglasDerivacion.add(reglaDerivacion); // Añadimos la regla al array

        derivacion = new Definicion(sustituto, reglasDerivacion); // Creamos la nueva definición

        creadoNuevo = true;
    }
}

```

Ilustración 21: Método sustituyeSimb() - 1

```

// Buscamos al sustituido en la lista de símbolos de la regla y lo cambiamos por sustituto
for (int i = 0; i<simbolos.size(); i++) {
    if (simbolos.get(i).equals(sustituido)) {
        simbolos.remove(i);
        simbolos.add(i, sustituto);
    }
}

if (creadoNuevo == true) {
    // Si se ha creado una nueva definición, se añade a la gramática
    defs.add(derivacion);
}
}

```

Ilustración 22: Método sustituyeSimb() - 2

Para el siguiente método hemos creado una variable nueva en la clase “**Regla**”, es una variable estática que nos permite llevar un mismo contador para todos los objetos de tipo de “**Regla**”, de esta forma cada vez que creamos un nuevo símbolo no terminal no tendremos problemas con que se repita el nombre de Aux1, Aux2, ...

Esta variable se obtiene mediante el método estático *getContador()* y se actualiza mediante el método estático *setContador()*.

```

/**
 * Crea un símbolo no terminal auxiliar nuevo con el nombre "Aux" + contador
 * @return El símbolo no terminal nuevo creado
 */
public static NTSymbol creaSimbolo() {

    String nombreSimb = "Aux" + Regla.getContador();
    Regla.setContador(Regla.getContador()+1);

    NTSymbol auxSimbol = new NTSymbol(nombreSimb);

    return auxSimbol;
}

```

Ilustración 23: Método creaSimbolo()

```
/**
 * Cuenta el número de símbolos terminales que hay en una regla
 * @return el número de símbolos terminales que tiene la regla
 */
public int getNumTerminales() {
    int nTerminales = 0;

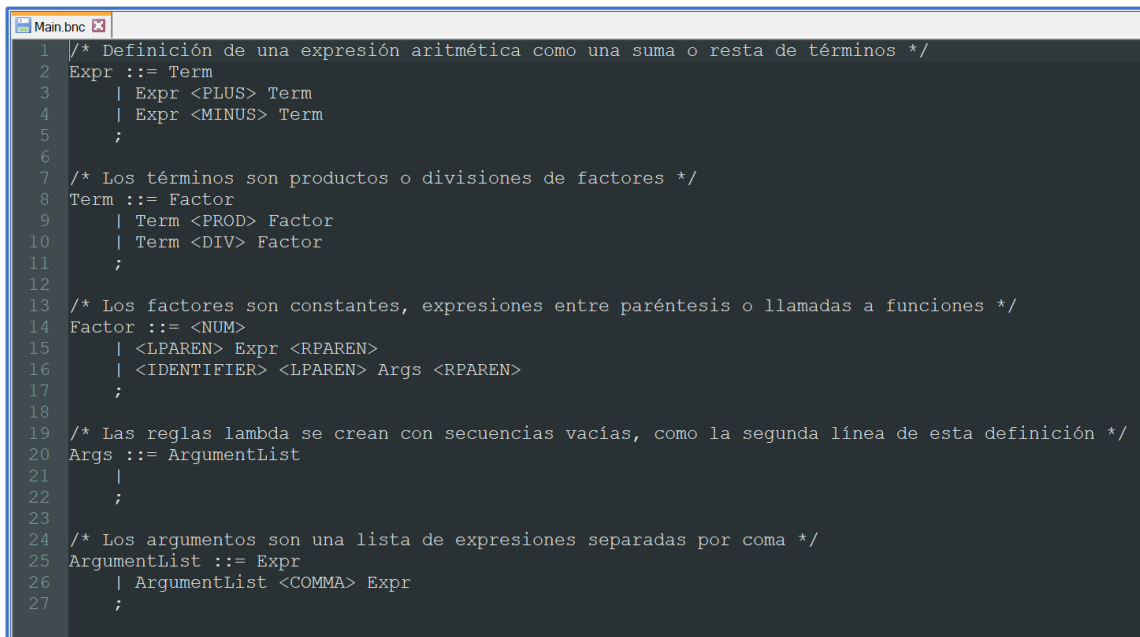
    // Recorremos la lista de símbolos, contando los que sean terminales
    for (Symbol sActual : simbolos)
        if (sActual instanceof TSymbol) nTerminales++;    // Si el símbolo es terminal, lo contamos

    return nTerminales;
}
```

Ilustración 24: Método getNumTerminales()

Pruebas sobre la aplicación

Para comprobar el correcto funcionamiento de la aplicación hemos tomado el fichero de entrada proporcionado junto al enunciado de la práctica.



```
1 /* Definición de una expresión aritmética como una suma o resta de términos */
2 Expr ::= Term
3       | Expr <PLUS> Term
4       | Expr <MINUS> Term
5       ;
6
7 /* Los términos son productos o divisiones de factores */
8 Term ::= Factor
9       | Term <PROD> Factor
10      | Term <DIV> Factor
11      ;
12
13 /* Los factores son constantes, expresiones entre paréntesis o llamadas a funciones */
14 Factor ::= <NUM>
15         | <LPAREN> Expr <RPAREN>
16         | <IDENTIFIER> <LPAREN> Args <RPAREN>
17         ;
18
19 /* Las reglas lambda se crean con secuencias vacías, como la segunda línea de esta definición */
20 Args ::= ArgumentList
21       |
22       ;
23
24 /* Los argumentos son una lista de expresiones separadas por coma */
25 ArgumentList ::= Expr
26               | ArgumentList <COMMA> Expr
27               ;
```

Ilustración 25: Ejemplo fichero de entrada

Si todo va bien, al ejecutar el traductor junto al archivo de entrada en el mismo directorio se generará una salida.

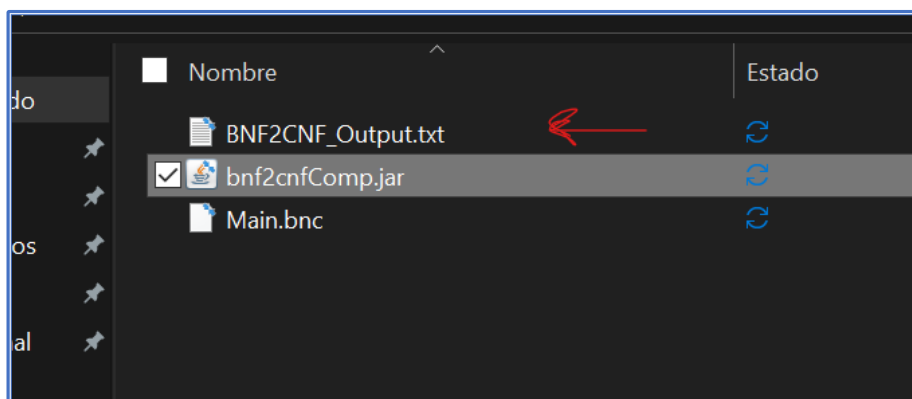


Ilustración 26: Salida del traductor (Explorador de archivos)

El resultado es algo similar a lo siguiente:

<pre> Expr ::= Expr Aux9 Expr Aux10 Term Aux11 Term Aux12 <NUM> Aux5 Aux13 Aux7 Aux14 Aux7 Aux15; Term ::= Term Aux11 Term Aux12 <NUM> Aux5 Aux13 Aux7 Aux14 Aux7 Aux15; Factor ::= <NUM> Aux5 Aux13 Aux7 Aux14 Aux7 Aux15; Args ::= ArgumentList Aux16 Expr Aux9 Expr Aux10 Term Aux11 Term Aux12 <NUM> Aux5 Aux13 Aux7 Aux14 Aux7 Aux15; </pre>	<pre> ArgumentList ::= ArgumentList Aux16 Expr Aux9 Expr Aux10 Term Aux11 Term Aux12 <NUM> Aux5 Aux13 Aux7 Aux14 Aux7 Aux15; Aux1 ::= <PLUS>; Aux2 ::= <MINUS>; Aux3 ::= <PROD>; Aux4 ::= <DIV>; Aux5 ::= <LPAREN>; Aux6 ::= <RPAREN>; Aux7 ::= <IDENTIFIER>; Aux8 ::= <COMMA>; Aux9 ::= Aux1 Term; Aux10 ::= Aux2 Term; Aux11 ::= Aux3 Factor; Aux12 ::= Aux4 Factor; Aux13 ::= Expr Aux6; Aux14 ::= Aux5 Aux17; Aux15 ::= Aux5 Aux6; Aux16 ::= Aux8 Expr; Aux17 ::= Args Aux6; </pre>
--	--

Ilustración 27: Salida del traductor

También nos interesa comprobar que la gestión de errores funciona sin ningún problema, modificaremos el fichero para comprobar que se genera una salida de errores.

```

14 Factor ::= <NUM>
15     | <LPAREN> Expr <RPAREN>
16     | <IDENTIFIER> <LPAREN> Args <RPAREN>
17     ;
18
19 /* Error de prueba */
20 <DIV> ::= Expr
21     ;
22
23 /* Las reglas lambda se crean con secuencia */
24 Args ::= ArgumentList
25     ;

```

Ilustración 28: Ejemplo de error en la entrada del traductor

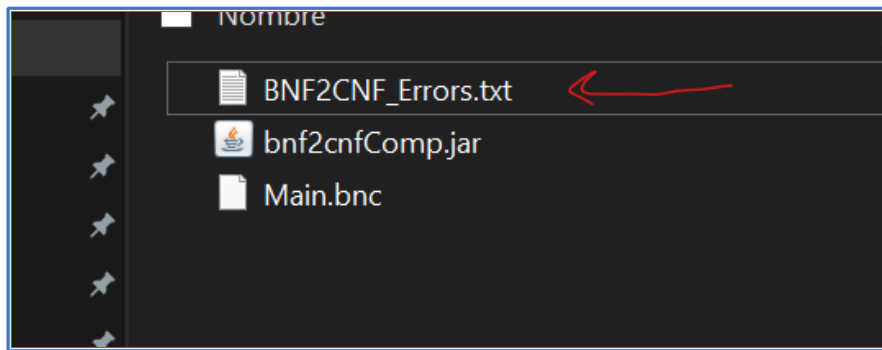


Ilustración 29: Salida de errores del traductor (Explorador de archivos)

El resultado obtenido es el fichero siguiente:

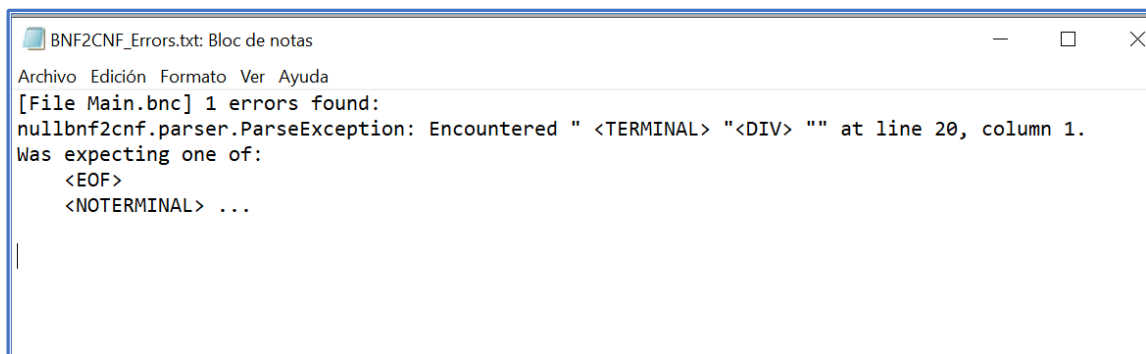


Ilustración 30: Salida de errores del traductor