GraphS

Graphs **S**pecification Language

Procesadores de Lenguajes

Curso 2008 – 2009

Jose Domingo López López
josed.lopez1@alu.uclm.es

Ángel Escribano Santamarina
angel.escribano1@alu.uclm.es

Escuela Superior de Informática
Universidad de Castilla-La Mancha

Índice

1. Introducción al problema

Se pretende diseñar una aplicación que permita al usuario especificar grafos mediante el lenguaje Graph**S** y realizar operaciones con dichos grafos.

Informalmente, un grafo es un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto.

Desde un punto de vista práctico, los grafos permiten estudiar las interrelaciones entre unidades que interactúan unas con otras. Por ejemplo, una red de computadoras puede representarse y estudiarse mediante un grafo, en el cual los vértices representan terminales y las aristas representan conexiones (las cuales, a su vez, pueden ser cables o conexiones inalámbricas).

Prácticamente cualquier problema puede representarse mediante un grafo, y su estudio transciende a las diversas áreas de las ciencias duras y las ciencias sociales.

La resolución del problema estará basada en un Procesador de Lenguajes que cogerá como entrada un fichero de texto, cuyo lenguaje será descrito a continuación, que definirá un conjunto finito de grafos sobre el cual se podrán aplicar una serie de algoritmos pare resolver problemas como el camino mínimo entre dos nodos y el árbol de recubrimiento mínimo a partir de un nodo.

El resultado será la visualización de una imagen correspondiente al resultado de dichos algoritmos.

2. Diseño del Procesador. Diagramas T

La aplicación será implementada en Java y tomará como entrada un fichero de texto escrito en el lenguaje de especificación de grafos Graph**S**. La salida será la visualización del grafo y el resultado de las operaciones.

Para abordar la resolución del problema se hace uso de otro procesador de lenguajes (compilador javac) que tendrá como entrada el código fuente de nuestra aplicación y devolverá el bytecode perteneciente a ésta. Dicho *bytecode* será ejecutado sobre una máquina virtual de Java (Java Virtual Machina) dando lugar a nuestra aplicación y permitiendo al usuario la visualización de los grafos que especifique. En la Figura 2.1 se puede observar como es el diagrama.

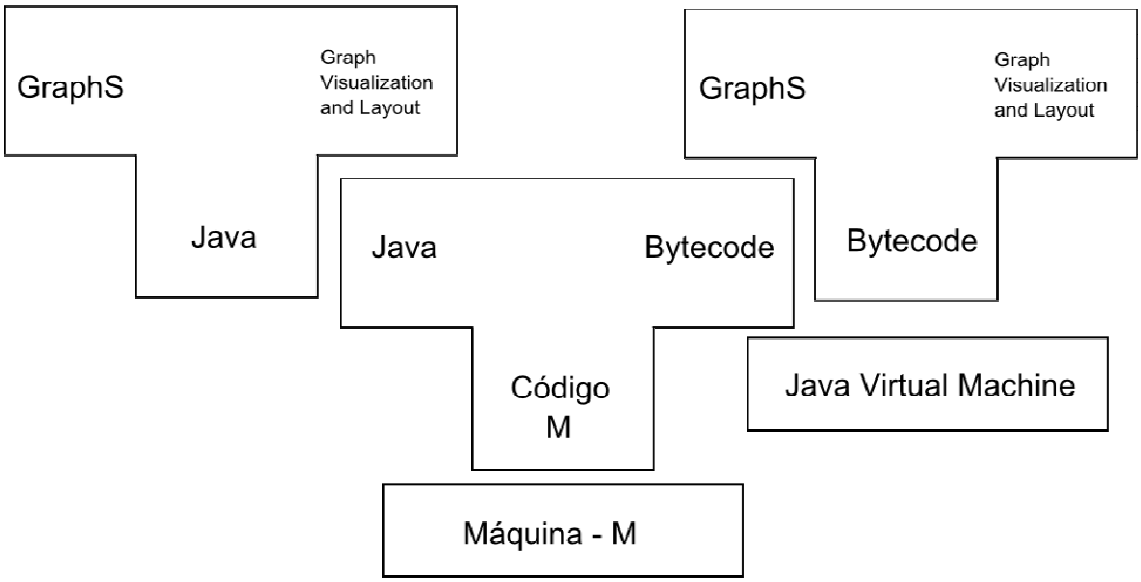

Figura 2.1: Diagramas T

3. Lenguaje de Entrada

El lenguaje Graph**S** consiste en dos secciones básicas: la primera es opcional y está destinada a la declaración de nodos y aristas globales que podrán ser utilizados por todos los grafos; la segunda es obligatoria y está destinada a la definición de grafos.

Un grafo es un bloque que comienza con la palabra reservada graph seguida de un identificador de grafo, y un cuerpo encerrado entre llaves. El cuerpo está compuesto por dos zonas: definición de grafo y definición de operaciones. Un grafo se puede definir mediante la declaración de sus nodos, aristas y conexiones o mediante una función de creación de grafos que implique a grafos ya definidos:

- Declaración de nodos: consta de una o más líneas que comienzan con la palabra reservada *node*, seguida de un identificador de nodo o una lista de identificadores de nodo separados por comas.
- Declaración de aristas: consta de una o más líneas que comienzan con la palabra reservada *edge,* seguida de un identificador de arista o una lista de identificadores de aristas separados por comas. Además, es posible caracterizar una arista con un peso. Éste se indicará con un número entero entre paréntesis después del identificador de la arista. En caso de no indicarse un peso, se asumirá que el peso de la arista es cero.
- Conexión de nodos: se representa mediante una asignación en la que la parte izquierda es un identificador de arista y la parte derecha son los dos identificadores de los nodos a unir con un símbolo entre ellos. Si queremos que la arista sea dirigida dicho símbolo será una flecha (->). Por el contrario si queremos que la arista sea no dirigida el símbolo será un guión (-).
- Definición de operaciones: consta de una sentencia para cada operación que comienza con la palabra reservada *op*. Los parámetros de las operaciones se indican entre paréntesis y separados por comas. Pueden ser operaciones que engloben varios grafos o que sólo hagan referencia a uno.

Cada sentencia termina en punto y coma.

3.1. Especificación EBNF

En la Figura 3.1 se puede estudiar la especificación EBNF correspondiente al lenguaje Graph**S** que lo describe formalmente:

```
SYNTAX ::= DECL GRAPHS_SET
GRAPHS_SET ::= GRAPH {GRAPH}
GRAPH ::= graph ID '{' BODY '}'
BODY ::= (DECL ARCS | OPS_GEN) [OPS]
DECL ::= {DECL_NODES | DECL_EDGES}
DECL_NODES ::= node ID {',' ID} ';'
DECL_EDGES ::= edge ID ['(' INT ')'] {',' ID ['(' INT
')']} ';'
ARCS ::= ARC {ARC}
ARC ::= ID '=' ID CONNECTOR ID ';'
CONNECTOR ::= '-' | '->'
OPS_GEN ::= OP_GEN
OP_GEN::= OP5
OPS ::= {OP}
OP ::= op (OP1 | OP2) ';'
OP1 ::= OPN1 '(' ID ')'
OPN1 ::= minimumSpanningTree
OP2 ::= OPN2 '(' ID ',' ID ')'
OPN2 ::= shortestPath
OP5 ::= OPN5 '(' ID ',' ID ',' ID ',' ID ',' ID ')'
OPN5 ::= union
ID ::= (MAYUS | MINUS) {MAYUS | MINUS | DIGITO}
INT ::= 0 | DIG {DIGITO}
MAYUS ::= A | B | C | D | E | F | G | H | I | J | K | L |
M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
MINUS ::= a | b | c | d | e | f | g | h | i | j | k | l |
m | n | o | p | q | r | s | t | u | v | w | x | y | z
DIG ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
DIGITO ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
COMMENT ::= '/''/' {^LINETERMINATOR} LINETERMINATOR
COMMENT_M ::= '/''*' {ANY_STRING | LINETERMINATOR} '*''/'
LINETERMINATOR ::= \r | \n | \r\n
```

Figura 3.1: Especificación EBNF

3.2. Semántica del lenguaje

Como se podrá observar en los ejemplos que se mostrarán a lo largo de este documento, se ha tratado en la medida de lo posible que la sintaxis de Graph**S** sea lo más parecida posible a la del lenguaje de programación C, de modo que la curva de aprendizaje sea ínfima. Del mismo modo, la semántica del lenguaje apenas aporta novedades con respecto a la de los lenguajes de programación habituales. A continuación se muestran algunos ejemplos:

- En un mismo ámbito, los identificadores de variables deben ser únicos.
- Un grafo es declarado como una variable global.
- Para utilizar una variable –un nodo, una arista o un grafo- ya sea en la conexión de aristas o en la definición de operaciones, ésta debe haber sido declarada anteriormente.
- Al conectar una arista es necesario que los nodos origen y destino se hayan declarado anteriormente como nodos, al igual que la arista, que debe haber sido declarada como arista.
- Si una arista es declarada sin peso, se asumirá que el peso es cero.
- Si un mismo identificador es utilizado en ámbito global y en ámbito local, prevalecerá el de ámbito local.

3.3. Diagramas de Conway

Los diagramas de Conway son unas estructuras sencillas que nos permiten comprender con facilidad un flujo. A continuación se mostrará un conjunto de diagramas que nos ayudarán a entender algunas de las líneas de la especificación EBNF mostrada en la Figura 3.1. Cada diagrama se corresponde con un *no-terminal*, representando los terminales en un círculo y los no-terminales en una caja rectangular o cuadrada.

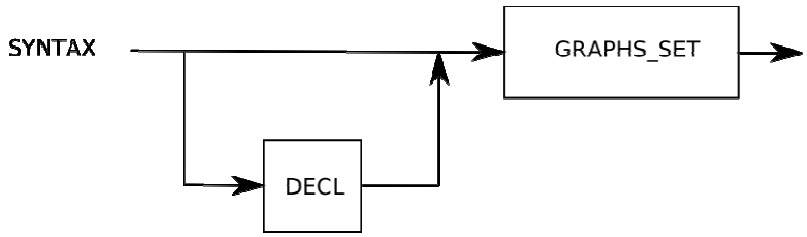

Figura 3.2: Diagrama de Conway para el no-terminal SYNTAX

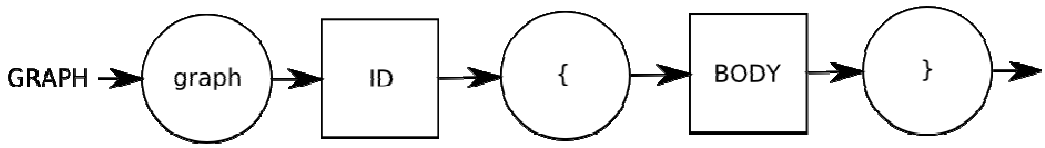

Figura 3.3: Diagrama de Conway para el no-terminal GRAPH

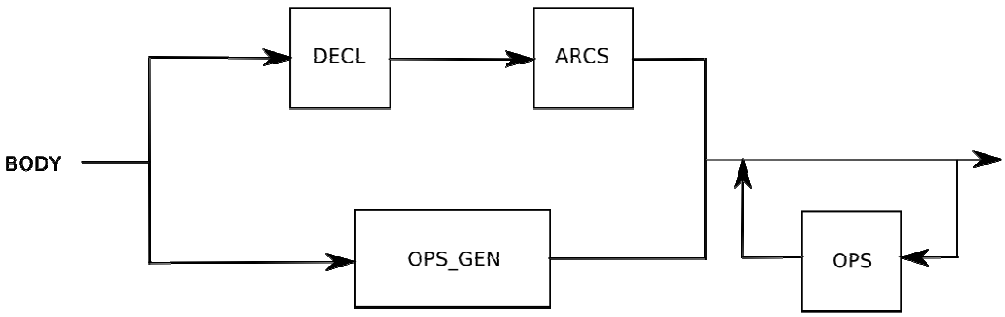

Figura 3.4: Diagrama de Conway para el no-terminal BODY

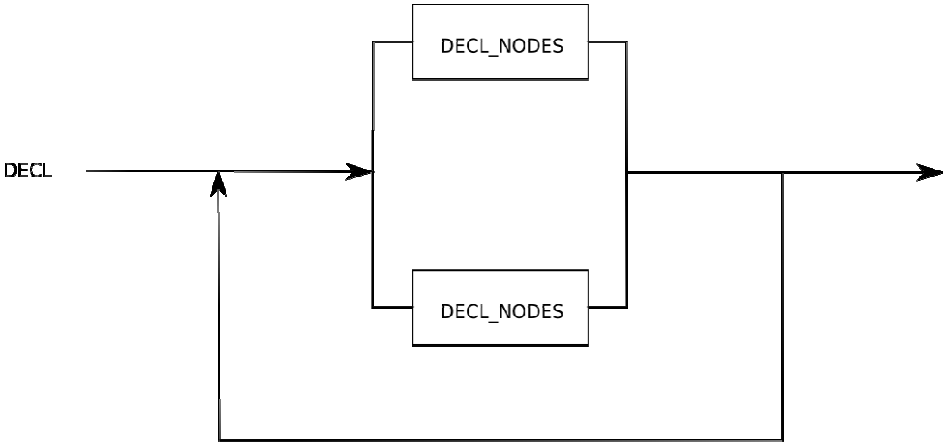

Figura 3.5: Diagrama de Conway para el no-terminal DECL

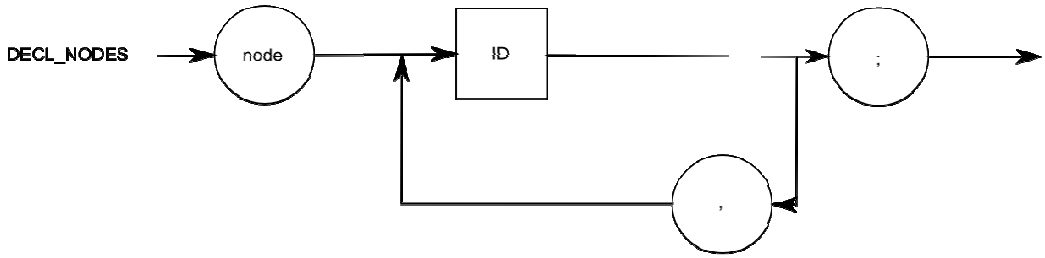

Figura 3.6: Diagrama de Conway para el no-terminal DECL_NODES

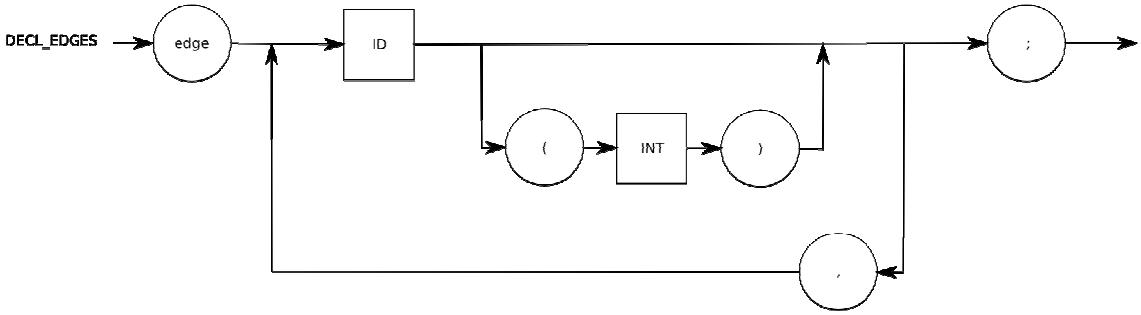

Figura 3.7: Diagrama de Conway para el no-terminal DECL_EDGES

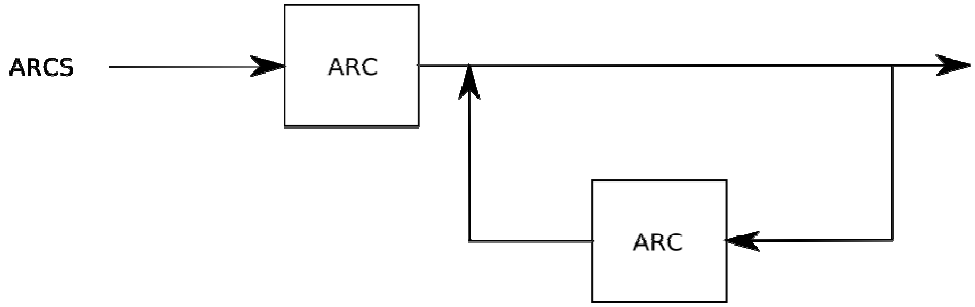

Figura 3.8: Diagrama de Conway para el no-terminal ARCS

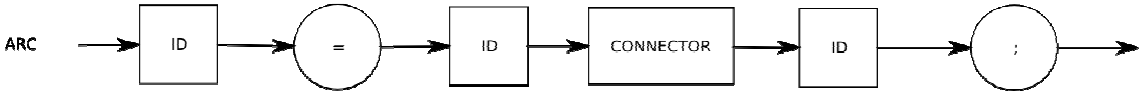

Figura 3.9: Diagrama de Conway para el no-terminal ARC

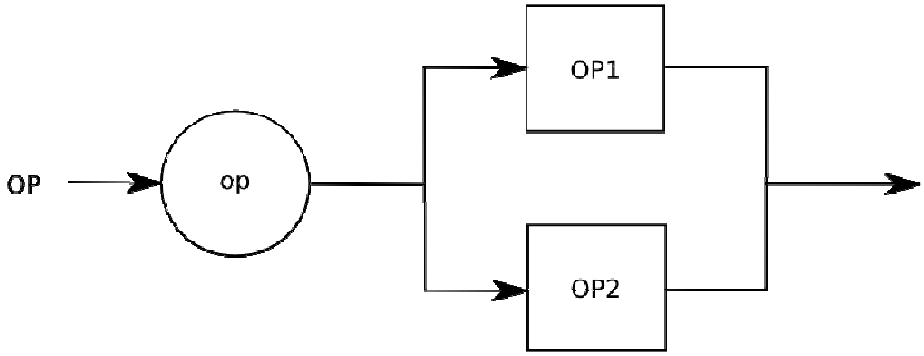

Figura 3.10: Diagrama de Conway para el no-terminal OP

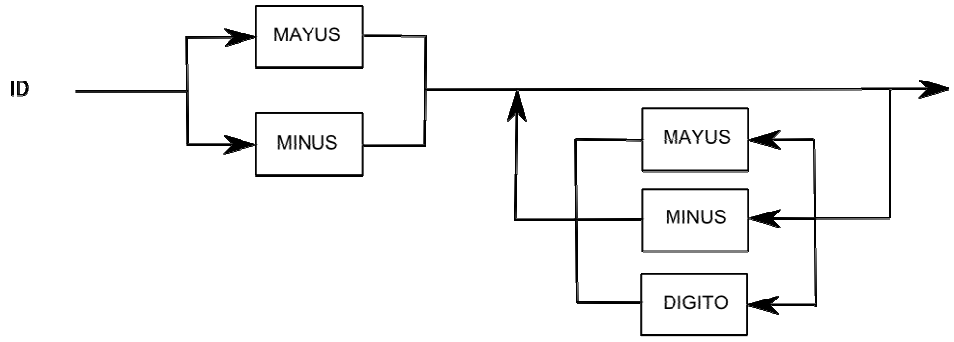

Figura 3.11: Diagrama de Conway para el no-terminal ID

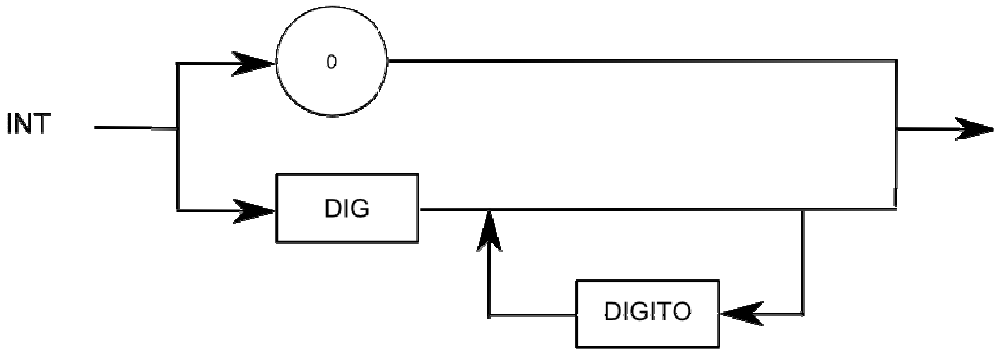

Figura 3.12: Diagrama de Conway para el no-terminal INT

3.4. Ejemplos de textos de entrada

3.4.1. Un grafo no dirigido y no valorado

```
/* Esto es un comentario */
//Esto también es un comentario

/* Ejemplo de un grafo no dirigido y no valorado */
graph Grafo1 {
     //Declaración de nodos
     node A, B, C, D;
     node E;
     //Declaración de aristas
     edge X, Y, Z, V, W;
     //no valoradas. Sería como poner X(0), Y(0)…
     edge M;

     //Conexión de nodos. El "-" indica una arista no
dirigida.
     X = A - B;
     Y = A - D;
     V = A - C;
     Z = B - D;
     W = C - D;
     M = C - E;

     //Definición de operaciones
     op shortestPath (A, E);
     op minimumSpanningTree (D);
}
```

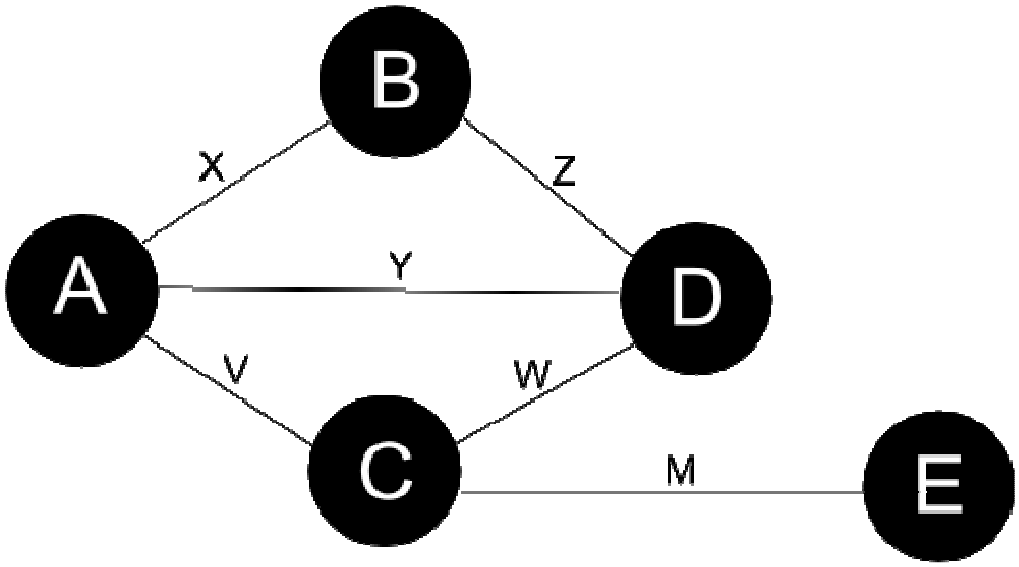

Figura 3.13: Ejemplo de grafo no dirigido y no valorado

3.4.2. Un grafo no dirigido y valorado

```
/* Ejemplo de un grafo no dirigido y valorado */
graph Grafo2 {
     node A, B, C;
     edge X(7), Y(9), Z(5);
     node D, E, F;
     edge V(3), W(2), M(1), N(6);

     X = A - B;
     Y = A - C;
     Z = A - D;
     V = D - C;
     W = D - F;
     M = B - E;
     N = F - E;

     op shortestPath (A, F);
     op minimumSpanningTree (E);
};
```

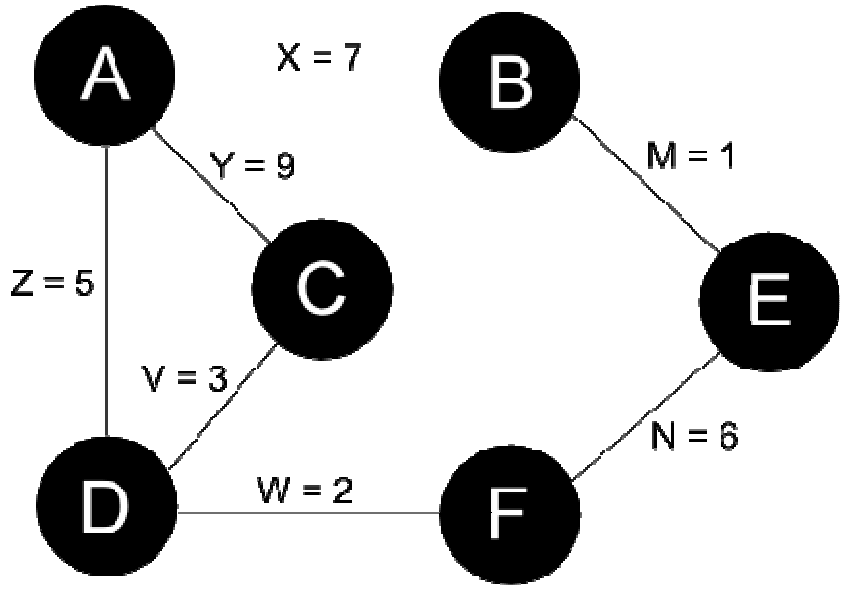

Figura 3.14: Ejemplo de grafo no dirigido y valorado

3.4.3. Un grafo dirigido y valorado

```
/* Ejemplo de un grafo dirigido y valorado */
graph Grafo3 {
     node A, B, C, D, E, F;
     edge X(7), Y(9), Z(5), V(3), W(2), M(1), N(6);

     X = A -> B; //"A -> B" indica una arista dirigida con
origen en A y destino en B
     Y = C -> A;
     Z = A -> D;
     V = D -> C;
     W = F -> D;
     M = B -> E;
     N = E -> F;

     op shortestPath (A, F);
     op minimumSpanningTree (E);
}
```

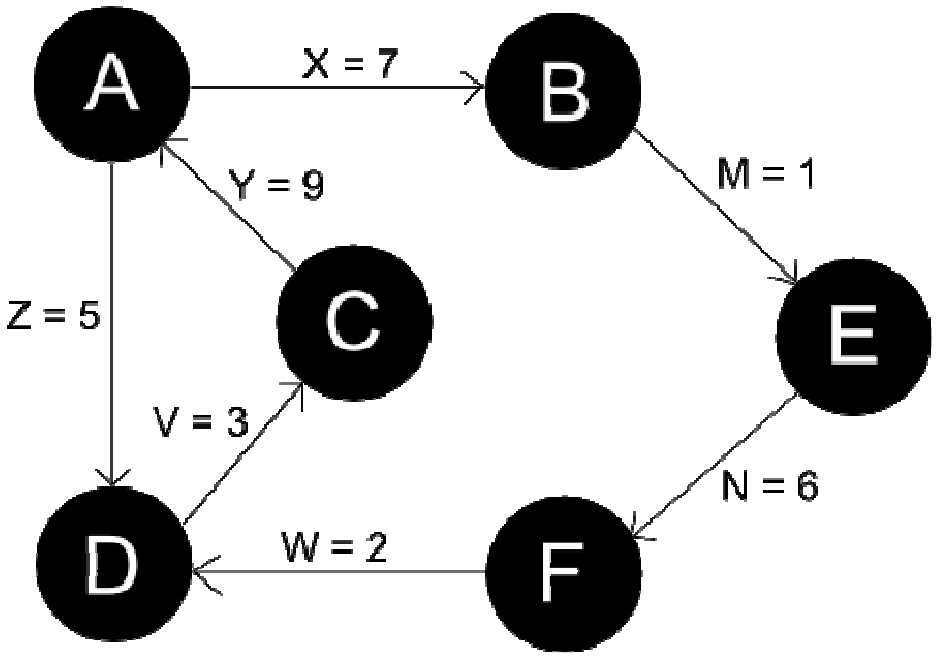

Figura 3.15: Ejemplo de grafo dirigido y valorado

3.4.4. Una unión de grafos no dirigidos y no valorados

```
node A, B, C, D;
edge X, Y, Z;

graph Grafo1 {
     node A, B;
     edge X;

     X = A - B;
}

graph Grafo2 {
     node C, D;
     edge Y;

     Y = C - D;
}

graph Grafo3 {
     union (Grafo1, Grafo2, B, C, Z);
}
```

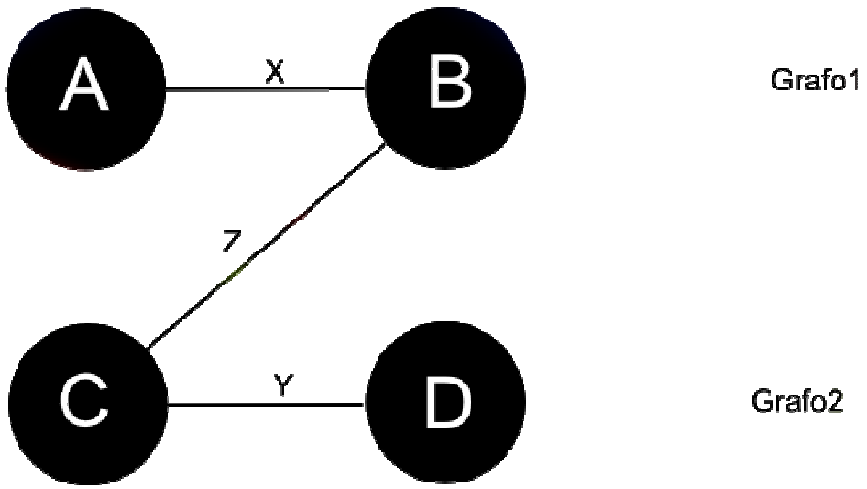

Figura 3.16: Ejemplo de unión de grafos

3.5. Alternativas de diseño

GraphS estará implementado en Java y será completamente portable a todo tipo de arquitecturas y sistemas operativos con una máquina virtual de Java instalada.

Hemos planteado dos alternativas a la hora de mostrar los resultados de las operaciones:

- Graphviz: librería de Java que genera imágenes svg de manera sencilla pero que no permiten la interacción del usuario con las partes del grafo. Se trata de un formato vectorial y libre.
- JGraph: librería de Java que permite la visualización y la interacción con cada una de las partes del grafo.

Uno de los problemas a los que nos enfrentamos a la hora de dibujar los grafos es la distribución de los nodos. La librería JGraph tiene un módulo que lo permite pero no es gratuito, mientras que GraphViz lo incluye en la librería libre. Por tanto, debido a esas ventajas y a su sencillez de uso a la hora de generar las imágenes nos hemos decantado por GraphViz.

4. Analizador Léxico

Una parte importante de la fase de análisis es el analizador léxico. De entre sus funciones destacan:

- Reconocer los símbolos o tokens que componen el texto fuente.
- Eliminar comentarios del texto fuente.
- Eliminar los espacios en blanco, saltos de línea y página, tabuladores,...
- Avisar de los errores léxicos detectados.
- Interacción con el analizador sintáctico:

Gracias a todo esto se simplifica el diseño del analizador sintáctico, ya que no tiene que tener en cuenta espacios en blanco, tabulaciones, espacios en blanco, etc.; y facilita la portabilidad, ya que las modificaciones del alfabeto de entrada pueden ser realizadas en el analizador léxico dejando al analizador sintáctico intacto.

4.1. Parte léxica del lenguaje

La parte léxica del lenguaje dado en el EBNF de la Figura 3.1, es dado por medio de la tabla de tokens (ver Figura 4.1), en la que se muestran los tokens, sus patrones y algunos lexemas.

Cabe destacar que dada la gran cantidad de palabras reservadas que puede llegar a tener el lenguaje, hemos optado por utilizar una tabla de palabras reservadas (ver Figura 4.2), la cual se consultará cada vez que reconozcamos el token *ident* (identificador). Esto se ha hecho para reducir el tamaño del autómata que reconoce los tokens del lenguaje.

Token	**Patrón**	**Lexema**	**Estado AFD**
graph	graph	graph	q12
node	node	node	q12
edge	edge	edge	q12
operadorUnario	minimumSpanningTree	minimumSpanningTree	q12
operadorBinario	shortestPath	shortestPath	q12
operadorQuinario	union	union	q12
l_bracket	{	{	q2
r_bracket	}	}	q3
l_paren	(	(	q4
r_paren	)	)	q5
semicolon	;	;	q6
comma	,	,	q7
equal	=	=	q8
connector	- | ->	-, ->	q9
ident	[a-zA-Z][[a-zA-Z0-9]*	A, arista1, nodoA	q12
number	0 | [1-9][0-9]*	0, 107, 991	q10 y q11

Figura 4.1: Tabla de tokens del lenguaje GraphS

Palabra reservada	Token
graph	graph
node	node
edge	edge
Op	op
minimumSpanningTree	operadorBinario
shortestPath	operadorBinario
union	operadorQuinario

Figura 4.2: Tabla de palabras reservadas del lenguaje GraphS

4.2. Autómata finito determinista

En la el autómata finito determinista (AFD) que reconoce los tokens del lenguaje Graph**S** y la asociación con cada token es mostrada en la tabla 2.

El estado *q12* representa cualquier secuencia de letras en minúscula y mayúscula. Cuando se llega a este estado, se debe comprobar si la secuencia leída corresponde con una de las palabras reservadas, almacenada en la tabla de palabras reservadas (ver Figura 4.2). Si es así, se devolverá el token correspondiente con la palabra reservada y el estado se considera final. Si no se encuentra en esa tabla, la secuencia que nos llevó a ese estado se considera un identificador (ident).

Los estados *q16, q19* y *q20* no son finales ya que se limitan a reconocer comentarios, espacios en blanco y caracteres de escape que no generan token y, por tanto, no se pasarán al analizador sintáctico.

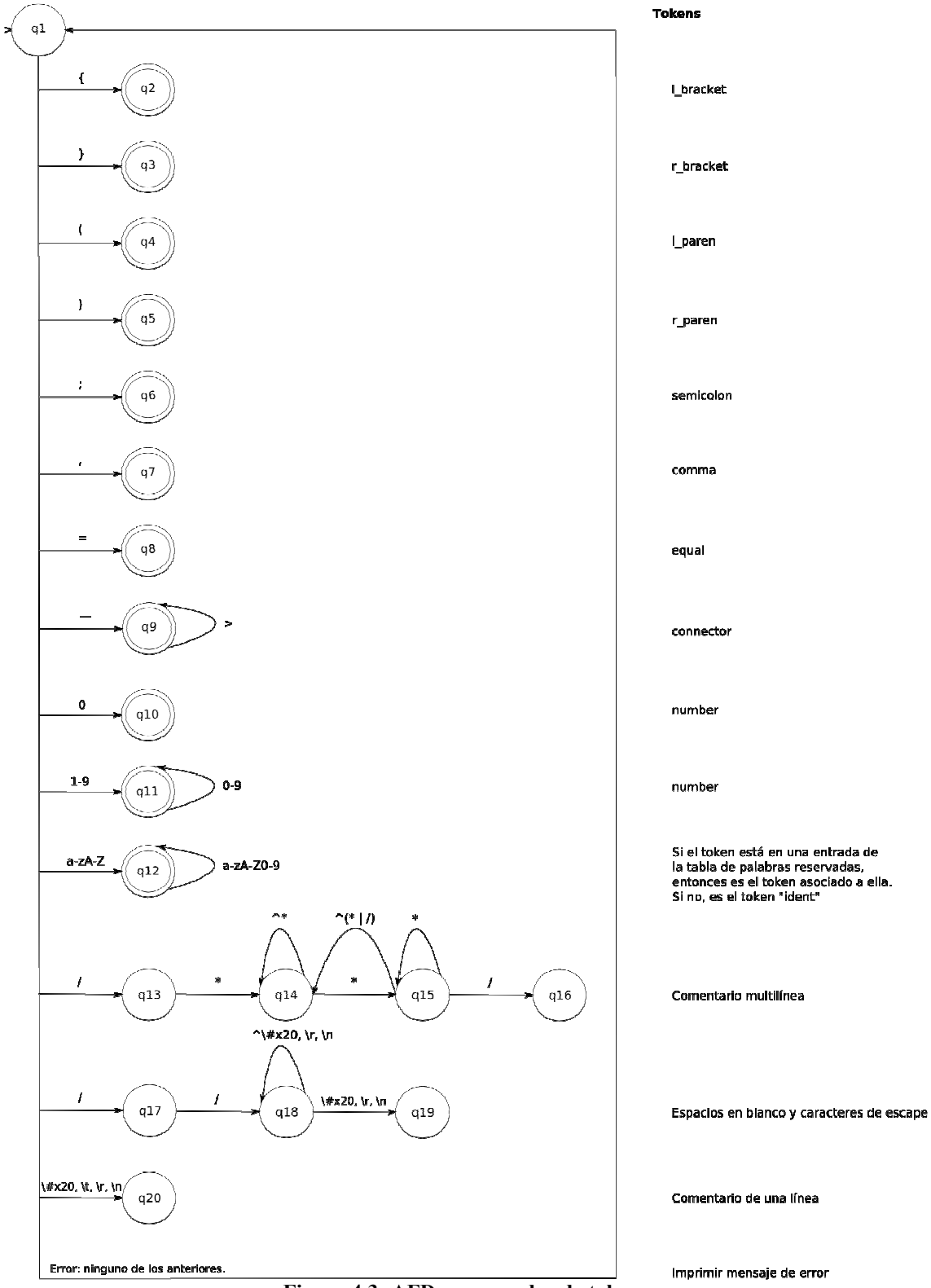

Figura 4.3: AFD reconocedor de tokens

4.3. Primera aproximación de AnaLex

En la se muestra una primera aproximación de lo que será el analizador léxico. Es una aproximación porque cuando se reconoce un token se muestra un mensaje informativo que indica el token reconocido en vez de crear un objeto de la clase *Symbol* y pasarlo al analizador sintáctico.

Cabe destacar también las siguientes características del analizador léxico (ver Figura 4.4):

- En la *Sección de código de usuario* se ha implementado una clase **Utility** en la cual mantendremos la tabla de palabras reservadas y una tabla de errores, así como las funciones correspondientes a la comprobación de una cadena en la tabla de palabras reservadas, devolver el token de una palabra reservada, e imprimir mensajes de error.
- En el *Sección de opciones y declaraciones* se observa que:
  - El nombre de la clase que se genera es AnaLex.
  - El analizador léxico es autónomo ya que hemos creado nuestro propio método main, por eso se ha comentado la propiedad standalone y se ha añadido una sentencia *type Object*.
  - Está implementado por medio de bloques *switch*.
  - Tiene dos estados adicionales para el tratamiento de comentarios mono-línea y multi-línea.
- En la *Sección de reglas léxicas* se puede observar la definición de los estados declarados en la sección anterior así como la definición del estado YYINITIAL. Es importante observar también la inclusión de reglas que reconocen cadenas no válidas, así como cierres de comentario sin apertura, e identificadores que comienzan por un dígito. El tratamiento apertura de comentarios sin cierre se hace en la sección de opciones y declaraciones en el bloque eof.

```
/* Codigo de usuario */

class Utility {
      public static final String Keywords[] =
{"graph","node","edge","op","minimumSpanningTree","shortestPath","unio
n"};
      public static final String errorMsg[] = {
            "Identificador no valido",
            "Error de sintaxis"
      };

      public static boolean isKeyword (String cadena) {
            boolean found = false;
            for (int i = 0; i < Keywords.length && !found; i++) {
                  if (cadena.equals(Keywords[i])) found = true;
            }
            return found;
      }

      public static String Keyword (String cadena) {
            String token = "";

            if (cadena.equals("graph")) token = "graph";
```

```
            else if (cadena.equals("node")) token = "node";
            else if (cadena.equals("edge")) token = "edge";
            else if (cadena.equals("op")) token = "op";
            else if (cadena.equals("minimumSpanningTree")) token =
"operadorUnario";
            else if (cadena.equals("shortestPath")) token =
"operadorBinario";
            else if (cadena.equals("union")) token =
"operadorQuinario";
            else System.out.println (":: ERROR. El lexema no se
encuentra asociado a ningun token.");

            return token;

      }

      public static String errorMsg (int error) {
            return errorMsg[error];
      }

}


class Main {
      public static void main (String argv[]) {
            if (argv.length == 0 || argv.length > 1) {
                  System.out.println("Uso : java AnaLex <inputfile>");
            }
            else {
                  AnaLex scanner = null;
                  try {
                        scanner = new AnaLex( new
java.io.FileReader(argv[0]));
                        System.out.println (":: Analizando fichero
"+argv[0]+"\n");
                        scanner.yylex();
                  }
                        catch (java.io.FileNotFoundException e) {
                        System.out.println("File not found :
\""+argv[0]+"\"");
                  }
                  catch (java.io.IOException e) {
                        System.out.println("IO error scanning file
\""+argv[0]+"\"");
                        System.out.println(e);
                  }
                  catch (Exception e) {
                        System.out.println("Unexpected exception:");
                        e.printStackTrace();
                  }
            }
      }
}


%%

/* Seccion de opciones y declaraciones */

%class AnaLex
/* %standalone */
```

```
%type Object
%line
%column
%switch
%states COMMENT, COMMENTM

%eof{
      if(zzLexicalState==COMMENTM)
            System.out.println(Utility.errorMsg(ERROR_SYNTAX)+".
Apertura de comentario sin cierre. Linea: "+(yyline+1)+" Columna:
"+(yycolumn+1));
%eof}

%{
public static final int ERROR_MSG_IDENT = 0;
public static final int ERROR_SYNTAX  = 1;



%}

/* Macros */

LineTerminator = \r | \n | \r\n
WhiteSpace = {LineTerminator} | [\t\f] | " "



%%


<YYINITIAL> {
      "//"  { yybegin(COMMENT); }
      "/*"  { yybegin(COMMENTM); }
      [a-zA-Z][a-zA-Z0-9]*    {
                              if (Utility.isKeyword(yytext())) {
                                    String Keyword = new String();
                                    Keyword =
Utility.Keyword(yytext());
                                    if (Keyword.equals(yytext())) {
                                          System.out.println ("Token
reconocido: "+Keyword+" Linea: " +(yyline+1)+" Columna:
"+(yycolumn+1));
                                    } else {
                                          System.out.println ("Token
reconocido: "+Keyword+" Lexema: "+yytext()+" Linea: " +(yyline+1)+"
Columna: "+(yycolumn+1));
                                    }
                              } else {
                                    System.out.println("Token
reconocido: ident Lexema: "+yytext()+" Linea: " +(yyline+1)+" Columna:
"+(yycolumn+1));
                              }
                        }
      "{"               { System.out.println("Token reconocido:
l_bracket Linea: " +(yyline+1)+" Columna: "+(yycolumn+1)); }
      "}"               { System.out.println("Token reconocido:
r_bracket Linea: " +(yyline+1)+" Columna: "+(yycolumn+1)); }
      "("               { System.out.println("Token reconocido: l_paren
Linea: " +(yyline+1)+" Columna: "+(yycolumn+1)); }
      ")"               { System.out.println("Token reconocido: r_paren
```

```
Linea: " +(yyline+1)+" Columna: "+(yycolumn+1)); }
      ";"               { System.out.println("Token reconocido:
semicolon Linea: " +(yyline+1)+" Columna: "+(yycolumn+1)); }
      ","               { System.out.println("Token reconocido: comma
Linea: " +(yyline+1)+" Columna: "+(yycolumn+1)); }
      "="               { System.out.println("Token reconocido: equal
Linea: " +(yyline+1)+" Columna: "+(yycolumn+1)); }
      "-" | "->"        { System.out.println("Token reconocido:
connector Linea: " +(yyline+1)+" Columna: "+(yycolumn+1)); }
      0                 { System.out.println("Token reconocido: number
Lexema: "+yytext()+" Linea: " +(yyline+1)+" Columna: "+(yycolumn+1));
}
      [1-9][0-9]*       { System.out.println("Token reconocido: number
Lexema: "+yytext()+" Linea: " +(yyline+1)+" Columna: "+(yycolumn+1));
}
      "*/"              {
System.out.println(Utility.errorMsg(ERROR_SYNTAX)+". Fin de comentario
sin apertura. Linea: "+(yyline+1)+" Columna: "+(yycolumn+1)); }
      [0-9][a-zA-Z0-9]* {
System.out.println(Utility.errorMsg(ERROR_MSG_IDENT)+" <"+yytext()+">
Linea: "+(yyline+1)+" Columna: "+(yycolumn+1)); }
      {WhiteSpace}            { }
      .                 { System.out.println("Expresion ilegal
<"+yytext()+"> Linea: "+(yyline+1)+" Columna: "+(yycolumn+1)); }

}



<COMMENT> {
      {LineTerminator}  { yybegin(YYINITIAL); }
      .     { }
}


<COMMENTM> {
      "*/"  { yybegin(YYINITIAL); }
      {LineTerminator}  { }
      .     { }
}
```

Figura 4.4: Primera aproximación de Analizador Léxico

4.4. Ejemplos de cadenas de entrada

A continuación se mostrarán dos ejemplos en los cuales se puede ver la cadena de entrada al analizador léxico, y la información que éste nos proporciona a la salida.

4.4.1. Ejemplo de cadena perteneciente al lenguaje

```
/* Ejemplo de un grafo no dirigido y no valorado */
graph Grafo1 {
     node A, B;
     edge X;

     X = A - B;

     op shortestPath (A, E);
     op minimumSpanningTree (D);
}
```

Figura 4.5: Entrada al analizador léxico. Cadena perteneciente al lenguaje

```
:: Analizando fichero Ejemplo1-pertenece.txt

Token reconocido: graph Linea: 2 Columna: 1
Token reconocido: ident Lexema: Grafo1 Linea: 2 Columna: 7
Token reconocido: l_bracket Linea: 2 Columna: 14
Token reconocido: node Linea: 3 Columna: 2
Token reconocido: ident Lexema: A Linea: 3 Columna: 7
Token reconocido: comma Linea: 3 Columna: 8
Token reconocido: ident Lexema: B Linea: 3 Columna: 10
Token reconocido: semicolon Linea: 3 Columna: 11
Token reconocido: edge Linea: 4 Columna: 2
Token reconocido: ident Lexema: X Linea: 4 Columna: 7
Token reconocido: semicolon Linea: 4 Columna: 8
Token reconocido: ident Lexema: X Linea: 6 Columna: 2
Token reconocido: equal Linea: 6 Columna: 4
Token reconocido: ident Lexema: A Linea: 6 Columna: 6
Token reconocido: connector Linea: 6 Columna: 8
Token reconocido: ident Lexema: B Linea: 6 Columna: 10
Token reconocido: semicolon Linea: 6 Columna: 11
Token reconocido: op Linea: 8 Columna: 2
Token reconocido: operadorBinario Lexema: shortestPath Linea: 8
Columna: 5
Token reconocido: l_paren Linea: 8 Columna: 18
Token reconocido: ident Lexema: A Linea: 8 Columna: 19
Token reconocido: comma Linea: 8 Columna: 20
Token reconocido: ident Lexema: E Linea: 8 Columna: 22
Token reconocido: r_paren Linea: 8 Columna: 23
Token reconocido: semicolon Linea: 8 Columna: 24
Token reconocido: op Linea: 9 Columna: 2
Token reconocido: operadorUnario Lexema: minimumSpanningTree
```

```
Linea: 9 Columna: 5
Token reconocido: l_paren Linea: 9 Columna: 25
Token reconocido: ident Lexema: D Linea: 9 Columna: 26
Token reconocido: r_paren Linea: 9 Columna: 27
Token reconocido: semicolon Linea: 9 Columna: 28
Token reconocido: r_bracket Linea: 10 Columna: 1
```

Figura 4.6: Salida informativa del analizador léxico. Cadena perteneciente al lenguaje

4.4.2. Ejemplo de cadena no perteneciente al lenguaje

```
/* Ejemplo de un grafo dirigido y valorado */
graph 999Grafo2 {
     node 0A, B0;
     edge X(7);

     X += A - B;

     op shortestPath (A, F);
     op minimumSpanningTree (E);
}
```

Figura 4.7: Entrada al analizador léxico. Cadena no perteneciente al lenguaje

```
:: Analizando fichero Ejemplo2-nopertenece.txt

Token reconocido: graph Linea: 2 Columna: 1
Identificador no valido <999Grafo2> Linea: 2 Columna: 7
Token reconocido: l_bracket Linea: 2 Columna: 17
Token reconocido: node Linea: 3 Columna: 2
Identificador no valido <0A> Linea: 3 Columna: 7
Token reconocido: comma Linea: 3 Columna: 9
Token reconocido: ident Lexema: B0 Linea: 3 Columna: 11
Token reconocido: semicolon Linea: 3 Columna: 13
Token reconocido: edge Linea: 4 Columna: 2
Token reconocido: ident Lexema: X Linea: 4 Columna: 7
Token reconocido: l_paren Linea: 4 Columna: 8
Token reconocido: number Lexema: 7 Linea: 4 Columna: 9
Token reconocido: r_paren Linea: 4 Columna: 10
Token reconocido: semicolon Linea: 4 Columna: 11
Token reconocido: ident Lexema: X Linea: 6 Columna: 2
Expresion ilegal <+> Linea: 6 Columna: 4
Token reconocido: equal Linea: 6 Columna: 5
Token reconocido: ident Lexema: A Linea: 6 Columna: 7
Token reconocido: connector Linea: 6 Columna: 9
Token reconocido: ident Lexema: B Linea: 6 Columna: 11
Token reconocido: semicolon Linea: 6 Columna: 12
Token reconocido: op Linea: 8 Columna: 2
Token reconocido: operadorBinario Lexema: shortestPath Linea: 8
Columna: 5
Token reconocido: l_paren Linea: 8 Columna: 18
Token reconocido: ident Lexema: A Linea: 8 Columna: 19
Token reconocido: comma Linea: 8 Columna: 20
```

```
Token reconocido: ident Lexema: F Linea: 8 Columna: 22
Token reconocido: r_paren Linea: 8 Columna: 23
Token reconocido: semicolon Linea: 8 Columna: 24
Token reconocido: op Linea: 9 Columna: 2
Token reconocido: operadorUnario Lexema: minimumSpanningTree
Linea: 9 Columna: 5
Token reconocido: l_paren Linea: 9 Columna: 25
Token reconocido: ident Lexema: E Linea: 9 Columna: 26
Token reconocido: r_paren Linea: 9 Columna: 27
Token reconocido: semicolon Linea: 9 Columna: 28
Token reconocido: r_bracket Linea: 10 Columna: 1
```

Figura 4.8: Salida informativa del analizador léxico. Cadena no perteneciente al lenguaje

5. Analizador Sintáctico

El analizador sintáctico es la siguiente fase en el proceso de análisis. Sus funciones básicas son las siguientes:

- Analizar la secuencia de tokens y verificar si son sintácticamente correctos.
- Obtener una representación interna del texto fuente.
- Avisar de los errores sintácticos detectados.

La construcción de esta parte se ha hecho en dos iteraciones: la primera para traducir la especificación EBNF a una gramática libre de contexto; y la segunda para añadir producciones de error que informan de errores sintácticos. En la Figura 5.1 se muestra el resultado de esta fase, donde es aconsejable prestar atención a las producciones de error. Éstas tratan de informar de los errores que se puedan producir a nivel de bloque, es decir, en la definición de la estructura grafo, en la zona de declaración de variables, en la zona de conexión de aristas y en la zona de definición de operaciones.

```
/* Especificaciones de package e imports */
import java_cup.runtime.*;

/* Componentes de codigo de usuario */
parser code {:
	public void report_error (String message, Object info) {
		StringBuffer m = new StringBuffer("Error");

		if (info instanceof java_cup.runtime.Symbol) {
			java_cup.runtime.Symbol s = ((java_cup.runtime.Symbol)
info);
			if (s.left >= 0) {
				m.append(" en la linea "+(s.left));
				if (s.right >= 0)
					m.append(", columna "+(s.right));
			}
		}
		m.append(" : "+message);
		System.err.println(m);
	}

	public void report_fatal_error(String message, Object info) {
		report_error(message,info);
		System.exit(1);
	}
:};


/* Lista de simbolos de la gramatica (terminales y no terminales) */
terminal graph, node, edge, op, operadorUnario, operadorBinario,
operadorQuinario;
terminal l_bracket, r_bracket, l_paren, r_paren;
terminal semicolon, comma, equal, connector, ident, number;
non terminal SYNTAX, GRAPHS_SET, GRAPH, BODY, DECL, DECL_NODES, DECL_EDGES;
non terminal IDENTS, IDENTS_NODE, IDENTS_SIMPLE, IDENTS_WEIGHTED, ARCS, ARC,
OPS_GEN, OPS, OP, OP1, OP2, OP5;

/* Declaraciones de precedencia */

/* Especificación de la gramatica */
SYNTAX		::=	GRAPHS_SET
		|	DECL GRAPHS_SET
```

```
                ;
GRAPHS_SET      ::= GRAPH
                |   GRAPH GRAPHS_SET
                ;
GRAPH           ::= graph ident l_bracket BODY r_bracket
                | error
                        {: parser.report_error("Error de sintaxis en la definicion
de grafo.",null); :}
                ;
BODY            ::= DECL ARCS OPS
                |   OPS_GEN   OPS
                | error
                        {: parser.report_error("Error de sintaxis en la zona de
declaracion.",null); :}
                ;
DECL            ::= DECL_NODES
                |   DECL_EDGES
                ;
DECL_NODES      ::= node IDENTS_NODE semicolon
                |   node IDENTS_NODE semicolon DECL
                ;
IDENTS_NODE     ::= ident
                |   ident comma IDENTS_NODE
                ;
DECL_EDGES      ::= edge IDENTS semicolon
                |   edge IDENTS semicolon DECL
                ;
IDENTS          ::= IDENTS_SIMPLE
                |   IDENTS_WEIGHTED
                ;
IDENTS_SIMPLE::= ident
                |   ident comma IDENTS
                ;
IDENTS_WEIGHTED     ::= ident l_paren number r_paren
                |   ident l_paren number r_paren comma IDENTS
                ;
ARCS            ::= ARC
                |   ARCS ARC
                ;
ARC             ::= ident equal ident connector ident semicolon
                | error
                        {: parser.report_error("Error de sintaxis en la zona de
conexion de aristas.",null); :}
                ;
OPS_GEN         ::= OP5 semicolon
                ;
OPS             ::= op OP
                |   OPS op OP
                ;
OP              ::= OP1 semicolon
                |   OP2 semicolon
                | error
                        {: parser.report_error("Error de sintaxis en la zona de
operaciones.",null); :}
                ;
OP1             ::= operadorUnario   l_paren ident r_paren
                ;
OP2             ::= operadorBinario  l_paren ident comma ident r_paren
                ;
OP5             ::= operadorQuinario l_paren ident comma ident comma ident comma
ident comma ident r_paren
                ;
```

Figura 5.1: Analizador sintáctico con producciones de error

6. Analizador Semántico

El analizador semántico se encuentra empotrado en el analizador sintáctico, por lo que ambas fases se realizarán al mismo tiempo. Las funciones del analizador semántico son las siguientes:

- Dar significado a las construcciones del lenguaje fuente (generación de código).
- Acabar de completar el lenguaje fuente ya que hay determinados aspectos del lenguaje que no pueden o es difícil de representar por medio de una gramática libre de contexto. Algunos ejemplos:
    - Que una variable sea declarada antes de ser usada.
    - Que cada variable se declare sólo una vez.
    - Que la sentencia de asignación tenga los tipos adecuados.
    - Que los parámetros de llamada a un procedimiento sean los requeridos en número y tipo.

6.1. Atributos de la gramática

A la hora de construir esta fase se han presentado ciertos problemas que su resolución es típica de atributos heredados -por ejemplo para añadir a la tabla de símbolos las variables con su ámbito correspondiente- pero esto supone una serie de conflictos cuando se está implementando un analizador sintáctico ascendente. La forma de resolverlos, es mediante el uso de marcadores, pero dada la complejidad de la gramática de GraphS, esto se puede convertir en un gran obstáculo. La forma para resolverlo ha sido mediante variables de control que se han implementado en la clase *parser*.

Aunque los atributos heredados hayan sido suprimidos, no significa que no se ejecuten reglas semánticas en el interior de las producciones. Esto es necesario hacerlo para que las variables de control que han sustituido a los atributos heredados tomen los valores necesarios cuando es requerido. Se deduce que si no hay atributos heredados, en este analizador únicamente encontraremos atributos sintetizados, los cuales son mostrados en la Figura 6.1.

Símbolo	Atributo	Significado
ident	String	Identificador de una variable.
connector	String	Tipo de conexión de una arista (dirigida o no dirigida).
number	Integer	Número que indica el peso de una arista.
OPS	ArrayList<Operation>	Lista de operaciones definidas en la zona de operaciones.
BODY	ArrayList<Operation>	Lista de operaciones definidas en la zona de operaciones.
OP	Operation	Operación definida en la zona de operaciones. Incluye el identificador y una lista de parámetros.

OP1	Operation	Operación definida en la zona de operaciones. Incluye el identificador y una lista de parámetros.
OP2	Operation	Operación definida en la zona de operaciones. Incluye el identificador y una lista de parámetros.
GRAPH	Graph	Estructura de tipo grafo que contiene un identificador, una lista de nodos, una lista de aristas y una lista de operaciones a realizar con él.
GRAPHS_SET	ArrayList<Graph>	Lista de grafos que se han definido el texto de entrada.

Figura 6.1: Atributos sintetizados del analizador semántico

6.2. Tabla de símbolos

Gracias a la tabla de símbolos nos es posible implementar una gran variedad de reglas semánticas con cierta facilidad. Sin ella, habría un uso excesivo de atributos heredados que habría que sustituir por marcadoresy complicaría muchísimo el diseño de esta fase.

La tabla de símbolos es una *tabla hash* que contiene tuplas <*clave,valor*>. En nuestro caso, la clave está compuesta por un identificador de variable y su ámbito (global o local), y el valor está compuesto por un objeto (la variable en cuestión) y un entero que indica las veces que ha sido utilizada. Tanto la clave como el valor se pueden encontrar en las clases ClaveTS.java y EntradaTS.java, dentro del paquete *analizador*.

Por otro lado, se han implementado una serie de operaciones realizar determinadas acciones en la tabla de símbolos tales como insertar, modificar, consultar y borrar ítems de la tabla de símbolos.

6.3. Reglas semánticas

Nos hemos visto en la necesidad de insertar reglas semánticas tanto al final de las producciones como en mitad de éstas. Como ya se ha comentado, las reglas semánticas que se ejecutan en mitad de las producciones son para ajustar los valores de variables de control, y las que se ejecutan al final de las producciones sirven para generar código que será subido hacia arriba cuando se reduzca y para hacer comprobaciones semánticas. Por simplicidad y para una mejor visualización, a continuación se explicarán lo que hacen las reglas semánticas de cada una de las producciones, pero dichas reglas no serán incluidas, por lo que se invita al lector a ver el fichero CUP del apéndice que se encuentra al final de este documento.

```
SYNTAX::= GRAPHS_SET:listaGrafos
```

Inicialmente el ámbito está con valor global. Esta producción contiene una regla semántica en el interior que indica que el ámbito de las declaraciones que se hagan será local, dado que no se harán declaraciones variables globales. Además, recibe una lista con los grafos que se han generado correctamente a partir del texto de entrada.

```
SYNTAX ::= DECL GRAPHS_SET:listaGrafos
```

Inicialmente el ámbito está con valor global. Esta producción contiene una regla semántica en el interior que indica que el ámbito de las declaraciones que se hagan será local, una vez que se han terminado de declarar las variables globales. Además, recibe una lista con los grafos que se han generado correctamente a partir del texto de entrada.

```
GRAPHS_SET ::= GRAPH:grafo
```

Esta producción es el caso base de la recursividad. Su regla semántica crea una lista de grafos y añade a ésta el grafo que recibe. Por último devuelve la lista hacia arriba.

```
GRAPHS_SET ::= GRAPH:grafo GRAPHS_SET:listaGrafos
```

Esta producción es el caso general de la recursividad. La regla semántica del interior reinicia las variables de control y limpia las variables de ámbito local de la tabla de símbolos, mientras que la regla del final realiza una operación similar al caso base (pero sin crear una nueva lista).

```
GRAPH ::= graph ident:identificador_grafo l_bracket
BODY:lista_operaciones r_bracket
```

Esta producción con tiene una regla semántica que comprueba que el identificador del grafo no existe en la tabla de símbolos. En dicho caso, genera un nuevo grafo con todos los nodos y aristas que hayan sido utilizados al menos una vez.

```
BODY ::= DECL ARCS OPS:lista_operaciones
```

Esta producción devuelve la lista de operaciones que se han especificado para este grafo.

```
BODY ::= OPS_GEN   OPS:lista_operaciones
```

Esta producción devuelve la lista de operaciones que se han especificado para este grafo. Nótese la diferencia con la producción anterior: en este caso el grafo ha sido creado mediante una operación generadora.

```
IDENTS_NODE ::= ident:identificador
IDENTS_SIMPLE    ::= ident:identificador
IDENTS_WEIGHTED    ::= ident:identificador l_paren number:peso r_paren
```

Estas producciones son casos bases de recursividades. Sus reglas semánticas comprueban que los identificadores no existen ya en la tabla de símbolos, en cuyo caso lo insertarán. En las producciones referentes a identificadores de aristas, si no se especifica un peso, se asumirá por defecto peso cero.

```
IDENTS_NODE ::= ident:identificador comma IDENTS_NODE
IDENTS_NODE ::= ident:identificador comma IDENTS
IDENTS_NODE ::= ident:identificador l_paren number:peso r_paren comma
IDENTS
```

Estas producciones son casos generales de recursividades. Sus reglas semánticas comprueban que los identificadores no existen ya en la tabla de símbolos, en cuyo caso lo insertarán. En las producciones referentes a identificadores de aristas, si no se especifica un peso, se asumirá por defecto peso cero.

```
ARC ::= ident:identificador_arista equal
ident:identificador_nodo_origen connector:conector
ident:identificador_nodo_destino semicolon
```

La regla semántica de esta producción comprobará que el primer identificador sea una variable declarada de tipo arista, y que el segundo y tercer identificador sean variables declaradas de tipo nodo. Además, comprobará que el tipo de arista sea coherente con el resto del grafo, es decir, que todas las aristas sean dirigidas o no dirigidas. Si todas las comprobaciones se llevan a cabo correctamente, los atributos *nodo origen* y *nodo destino* de la arista serán configurados, así como incrementado el contador de *veces utilizado*.

```
OPS ::= op OP:operación
```

Esta producción es el caso base de la recursividad y su regla semántica devuelve una lista de operaciones.

```
OPS ::= OPS:lista_operaciones op OP:operacion
```

Esta producción es el caso general de la recursividad y su regla semántica devuelve una lista de operaciones.

```
OP ::= OP1:operacion semicolonia
```

La regla semántica de esta producción devuelve una operación que más tarde será añadida a una lista de operaciones.

```
OP1 ::= operadorUnario:identificador_operacion   l_paren
ident:identificador_nodo r_paren
OP2 ::= operadorBinario:identificador_operacion l_paren
ident:identificador_nodo_origen comma ident:identificador_nodo_destino
r_paren
```

Las reglas semánticas de estas producciones se limitan a comprobar que existen los argumentos de las operaciones en la tabla de símbolos así como restricciones semánticas del tipo: para realizar el árbol de recubrimiento mínimo mediante el algoritmo de Kruskal, el grafo debe ser no-dirigido; y para realizar el camino mínimo entre dos nodos mediante Floyd, el grafo debe ser dirigido.

```
OP5 ::= operadorQuinario:identificador_operacion l_paren
ident:identificador_grafo_origen comma
ident:identificador_grafo_destino comma
ident:identificador_nodo_origen comma ident:identificador_nodo_destino
comma ident:identificador_arista r_paren
```

La regla semántica de esta producción que los dos grafos a partir de los cuales se generará el nuevo grafo existe en la tabla de símbolos y que los nodos origen y fin existen en dichos grafos. Por último, generará el grafo resultante de la operación generadora para insertarlo posteriormente en la tabla.

7. Generación de Código

A continuación se explicará brevemente en qué consiste cada una de las estructuras de datos que forman parte de la lógica de dominio de la aplicación.

7.1. Clase Node

Esta clase no es vital ya que a día de hoy la aplicación únicamente utiliza los identificadores de los nodos que son de tipo String. No obstante, se ha creado una clase que encapsula el identificador para que en un futuro se pueda añadir más información a los nodos.

```
public class Node implements Comparable<Node> {

    String name;
    boolean visited = false;   // used for Kosaraju's algorithm and
Edmonds's algorithm
    int lowlink = -1;          // used for Tarjan's algorithm
    int index = -1;            // used for Tarjan's algorithm

    public Node(String n){
        name = n;
    }

    public boolean equals (Object ob) {
        boolean equals = false;
        if ((ob != null) && (ob instanceof Node)) {
            Node n = (Node)ob;
            if (n.name.equals(name)) {
                equals = true;
            }
        }
        return equals;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int compareTo(Node n){
        if(n.equals(this))
            return 0;
        return -1;
    }

    public String toString(){
        String res = "";
        res += name;
        return res;
```

```
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return new Node(name);
    }
}
```

Figura 7.1: Clase Node

7.2. Clase Edge

Esta clase contiene toda la información relativa a las aristas que necesitaremos: nodo origen, nodo destino, identificador de arista y peso. Además, ha sido necesario implementar el método *compareTo* para poder ordenar las aristas en función de su peso (utilizado en el árbol de recubrimiento mínimo).

```
public class Edge implements Comparable<Edge> {

    String name;
    Node from, to;
    int weight;

    public Edge(String n, int w){
            name = n;
            from = null;
            to = null;
            weight = w;
    }

    public Edge(String n, Node f, Node t, int w){
            name = n;
            from = f;
            to = t;
            weight = w;
    }

    public Node getFrom() {
        return from;
    }

    public void setFrom(Node from) {
        this.from = from;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Node getTo() {
        return to;
    }

    public void setTo(Node to) {
```

```
        this.to = to;
    }

    public int getWeight() {
        return weight;
    }

    public void setWeight(int weight) {
        this.weight = weight;
    }

    public int compareTo(Edge e){
            return weight - e.weight;
    }

    public boolean equals (Object ob) {
            boolean equals = false;
            if ((ob != null) && (ob instanceof Edge)) {
                    Edge e = (Edge)ob;
                    if (from.equals(e.from) && to.equals(e.to)){
                            equals = true;
                    }
            }
            return equals;
    }

    public String toString(){
            String res = "";
            res += "Arista " + name + " desde "+from.toString()+"
hasta "+to.toString()+" con peso "+weight;
            return res;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        return new Edge (name, from, to, weight);
    }
}
```

Figura 7.2: Clase Edge

7.3. Clase Operation

Esta clase está compuesta por un identificador de operación y una lista genérica de objetos en los cuales se encontrarán insertados cada uno de los parámetros de la aplicación.

```
import java.util.ArrayList;

public class Operation {
    private String operation;
    private ArrayList<Object> params;

    public Operation(String operation, ArrayList<Object> params) {
        this.operation = operation;
        this.params = params;
    }
```

```
    public ArrayList<Object> getParams() {
        return params;
    }

    public void setParams(ArrayList<Object> params) {
        this.params = params;
    }

    public String getOperation() {
        return operation;
    }

    public void setOperation(String operation) {
        this.operation = operation;
    }


}
```

Figura 7.3: Clase Operation

7.4. Clase Graph

Esta clase encapsula todas las clases anteriores. Un grafo se compone de un identificador, unos atributos que indican si es dirigido y/o valorado, una lista de nodos, una lista de aristas y una lista de operaciones que ejecutaremos sobre sí mismo. Las operaciones que han sido implementadas son el camino mínimo entre dos nodos, el árbol de recubrimiento mínimo, y la unión de dos grafos.

```
import java.util.ArrayList;


public class Graph {
    String name;
    boolean weighted;
    boolean directed;
    ArrayList<Node> nodes;
    ArrayList<Edge> edges;
    ArrayList<Operation> operations;

    public Graph (String na, boolean w, boolean d, ArrayList<Node> n,
ArrayList<Edge> e) {
        name = na;
        weighted = w;
        directed = d;
        nodes = n;
        edges = e;
        operations = null;
    }

    public Graph (String na, boolean w, boolean d, ArrayList<Node> n,
ArrayList<Edge> e, ArrayList<Operation> op) {
        name = na;
        weighted = w;
        directed = d;
        nodes = n;
        edges = e;
```

```
        operations = op;
    }

    public String getName() {
        return name;
    }

    public ArrayList<Edge> getEdges() {
        return edges;
    }

    public ArrayList<Node> getNodes() {
        return nodes;
    }

    public boolean isDirected() {
        return directed;
    }

    public boolean isWeighted() {
        return weighted;
    }

    public ArrayList<Operation> getOperations() {
        return operations;
    }

    public void setOperations(ArrayList<Operation> operations) {
        this.operations = operations;
    }

    public String toString(){
        String res = "";

        res += name + "\n" + "valorado: " + weighted + "\n" +
"dirigido: " + directed + "\n";
        for (Node n: nodes) {
                res += n.toString() + "\n";
        }
        for (Edge e: edges) {
                res += e.toString() + "\n";
        }

        return res;
    }

    public boolean hasNode (String id) {
        boolean exists = false;

        Node n = new Node (id);
        int index = nodes.indexOf(n);
        if (index != -1) exists = true;

        return exists;
    }

    public Node getNode (String id) {
        Node n = new Node (id);
        int index = nodes.indexOf(n);

        return nodes.get(index);
```

```
    }

    public Edge getEdge (Node source, Node target) {
        Edge e = new Edge ("", source, target, 0);
        int index = edges.indexOf(e);

        return edges.get(index);
    }

    public Graph Kruskal () {
        Graph res = null;

        Kruskal kru = new Kruskal();
        res = kru.getMST(this);

        return res;
    }

    public Graph Floyd (Node source, Node target) {
        Graph res = null;

        Floyd flo = new Floyd();
        flo.calcShortestPaths(this);
        res = flo.getShortestPath(this, source, target);

        return res;
    }

    public static Graph Union (Graph graph1, Graph graph2, Node
source, Node target, Edge ed) {
        Graph res = null;
        String union_name = graph1.name + graph2.name;
        boolean weighted = false;
        if (graph1.weighted && graph2.weighted) weighted = true;
        boolean directed = false;
        if (graph1.directed && graph2.directed) directed = true;
        ArrayList<Node> nodes = new ArrayList<Node>();
        ArrayList<Edge> edges = new ArrayList<Edge>();
        for (Node n: graph1.nodes) {
                nodes.add(new Node(n.name + graph1.name));
        }
        for (Node n: graph2.nodes) {
                nodes.add(new Node(n.name + graph2.name));
        }
        for (Edge e: graph1.edges) {
                edges.add(new Edge(e.name + graph1.name, new
Node(e.from.name + graph1.name), new Node(e.to.name + graph1.name),
e.weight));
        }
        for (Edge e: graph2.edges) {
                edges.add(new Edge(e.name + graph2.name, new
Node(e.from.name + graph2.name), new Node(e.to.name + graph2.name),
e.weight));
        }
        // Metemos la arista que une los dos grafos
        edges.add(new Edge(ed.name + union_name, new Node(source.name
+ graph1.name), new Node(target.name + graph2.name), ed.weight));
        res = new Graph (union_name, weighted, directed, nodes,
edges);
        return res;
    }
```

Figura 7.4: Clase Graph

7.5. Clase Kruskal

Esta clase contiene el algoritmo que genera el árbol de recubrimiento mínimo de un grafo no dirigido. Para ello, ordena las aristas en orden ascendente en términos de peso y las va añadiendo a un nuevo grafo mientras que éstas no vayan creando ciclos.

```
import java.util.ArrayList;


public class Kruskal {

    /**
     * ALGORITMO DE KRUSKAL
     * ====================
     * Kruskal's algorithm finds a minimum spanning tree for a
connected, weighted, undirected graph.
     *
     * REFERENCIAS
     * ===========
     * http://algowiki.net/wiki/index.php/Kruskal%27s_algorithm
     * http://algowiki.net/wiki/index.php/Disjoint-set
     * http://algowiki.net/wiki/index.php/Edge
     * http://algowiki.net/wiki/index.php/Node
     */

    public Graph getMST(Graph graph){
        Graph res = new Graph (graph.name + "_Kruskal",
graph.weighted, graph.directed, new ArrayList<Node>(), new
ArrayList<Edge>());
        ArrayList<Edge> edges = new ArrayList<Edge>(graph.edges);
        java.util.Collections.sort(edges);
        DisjointSet<Node> nodeset = new DisjointSet<Node>();
        nodeset.createSubsets(graph.nodes);
        for(Edge e : edges){
            if(nodeset.find(e.from) != nodeset.find(e.to)){
                nodeset.merge(nodeset.find(e.from),
nodeset.find(e.to));
                res.edges.add(e);
            }
        }
    if (res.edges.size() > 0) res.nodes.addAll(graph.nodes);
        return res;
    }
}
```

Figura 7.5: Clase Kruskal

7.6. Clase Floyd

Esta clase contiene dos operaciones: la primera se encarga de calcular todas las distancias mínimas que hay entre cada uno de los nodos del grafo, añadiendo cual es el siguiente nodo que debería visitar para obtener ese coste; la segunda es una función recursiva que reconstruye el camino mínimo entre un par de nodos dado.

```
import java.util.ArrayList;
import java.util.Arrays;


public class Floyd {
    // The Floyd-Warshall algorithm finds the shortest path in a
weighted, directed graph for all node pairs in a single execution.
    static int[][] D;
    static Node[][] P;

    public void calcShortestPaths(Graph graph) {
        ArrayList<Node> nodes = graph.nodes;
        ArrayList<Edge> edges = graph.edges;

        D = initializeWeight(nodes, edges);
        P = new Node[nodes.size()][nodes.size()];

        for(int k=0; k<nodes.size(); k++){
            for(int i=0; i<nodes.size(); i++){
                for(int j=0; j<nodes.size(); j++){
                    if(D[i][k] != Integer.MAX_VALUE && D[k][j] !=
Integer.MAX_VALUE && D[i][k]+D[k][j] < D[i][j]){
                        D[i][j] = D[i][k]+D[k][j];
                        P[i][j] = nodes.get(k);
                    }
                }
            }
        }
    }

    public int getShortestDistance(ArrayList<Node> nodes, Node source,
Node target){
        int nSource = nodes.indexOf(source);
        int nTarget = nodes.indexOf(target);
        return D[nSource][nTarget];
    }

    public Graph getShortestPath(Graph graph, Node source, Node
target){
        ArrayList<Node> nodes = graph.nodes;
        int nSource = nodes.indexOf(source);
        int nTarget = nodes.indexOf(target);
        Graph res = new Graph (graph.name + "_Floyd", graph.weighted,
graph.directed, new ArrayList<Node>(), new ArrayList<Edge>());

        if(D[nSource][nTarget] == Integer.MAX_VALUE){
            return new Graph (graph.name + "_Floyd", graph.weighted,
graph.directed, new ArrayList<Node>(), new ArrayList<Edge>());
        }
        res.nodes = getIntermediatePath(graph, source, target);
        res.nodes.add(0, source);
        res.nodes.add(target);
        for (int i = 0; i < res.nodes.size()-1; i++) {
            res.edges.add(graph.getEdge(res.nodes.get(i),
res.nodes.get(i+1)));
        }
        return res;
    }

    private ArrayList<Node> getIntermediatePath(Graph graph, Node
source, Node target){
```

```
        ArrayList<Node> nodes = graph.nodes;
        int nSource = nodes.indexOf(source);
        int nTarget = nodes.indexOf(target);
        if(D == null){
            throw new IllegalArgumentException("Must call
calcShortestPaths(...) before attempting to obtain a path.");
        }
        if(P[nSource][nTarget] == null){
            return new ArrayList<Node>();
        }
        ArrayList<Node> path = new ArrayList<Node>();
        path.addAll(getIntermediatePath(graph, source,
P[nSource][nTarget]));
        path.add(P[nSource][nTarget]);
        path.addAll(getIntermediatePath(graph, P[nSource][nTarget],
target));
        return path;
    }

    private int[][] initializeWeight(ArrayList<Node> nodes,
ArrayList<Edge> edges){
        int nFrom = -1;
        int nTo = -1;
        int[][] Weight = new int[nodes.size()][nodes.size()];
        for(int i=0; i<nodes.size(); i++){
            Arrays.fill(Weight[i], Integer.MAX_VALUE);
        }
        for(Edge e : edges){
            nFrom = nodes.indexOf(e.from);
            nTo = nodes.indexOf(e.to);
            Weight[nFrom][nTo] = e.weight;
        }
        return Weight;
    }
}
```

Figura 7.6: Clase Floyd

8. Manual de usuario

El manual de usuario de la aplicación contiene toda la información relativa a la instalación y uso de la aplicación, así como ficheros de ejemplo. Dada su extensión no ha sido incluido en este documento pero puede encontrarse en este mismo directorio en formato CHM (Compiled HTML Help) o, en formato HTML accediendo desde el menú de ayuda de la aplicación.

9. Conclusiones

Después de la realización de esta práctica, se deduce que un procesador de lenguajes es una herramienta muy potente y versátil que se puede utilizar como solución apropiada para multitud de problemas en diferentes ámbitos.

La separación de las fases de análisis y de síntesis, así como las diferentes etapas que las conforman (analizadores léxico, sintáctico y semántico) facilitan mucho la tarea de implementar el procesador. Esto último también permite el implementar las etapas de forma consecutiva e incremental, y realizar pruebas para verificar su completitud y corrección. Para esto, v arias de las técnicas vistas en clase nos han sido de gran utilidad a la hora de realizar la práctica. Algunos ejemplos son la utilización de una tabla de símbolos para las comprobaciones semánticas en la traducción dirigida por la sintaxis, el uso de producciones de error para la recuperación de errores en el análisis sintáctico, entre otros.

Ciertos aspectos de la especificación de las características del lenguaje hecha por nosotros o de las modificaciones propuestas por el profesor, han conllevado mucha más dificultad de la esperada. Ejemplos de ello son:

- La declaración de variables de ámbito global, y la comprobación de conflictos con las locales a los grafos.
- La implementación de operaciones generadoras a partir de grafos ya definidos.
- La interfaz gráfica es un aspecto a cuidar en cualquier aplicación, la cual debe ser intuitiva y de fácil manejo.

En la misma línea deben ir los mensajes de error de cualquier tipo (léxicos, sintácticos y semánticos) mostrados al usuario, ya que deben ser lo más concretos y descriptivos posibles para facilitarle su localización y rectificación.

Un último aspecto a destacar, aunque no menos importante, es el estricto control de versiones que hemos llevado, gracias al cual en más de una ocasión hemos podido recuperar archivos accidentalmente eliminados, restaurar versiones correctas después de que alguna modificación produjera comportamientos inesperados, por no hablar de que utilizando este repositorio se evita trabajar con archivos desactualizados y la

sincronización del trabajo realizado por los distintos miembros del grupo se hace de una forma transparente.

Concretamente, el repositorio utilizado ha sido un repositorio subversion facilitado por google code, el cual se puede encontrar en la siguiente URL:

http://code.google.com/p/pl-graphs/

10. Futuras mejoras

En principio la interfaz gráfica ha sido desarrollada en Swing. No obstante, resultaría interesante la realización de una interfaz gráfica Web para que los usuarios no tengan que instalar ningún software adicional y puedan utilizar la herramienta por medio de un navegador tradicional como Internet Explorer, Opera, Mozilla Firefox, Safari, etc.

Otra posible mejora para la práctica actual sería introducir un parser que detectara la gramática del lenguaje GraphS y permitiera mostrar las distintas palabras reservadas del lenguaje de un color diferente, al igual que hacen la mayoría de IDEs de cualquier lenguaje de programación. Esto sería posible con el javabean disponible http://syntax.jedit.org/

Este editor es libre y se puede modificar, por lo que lo único que habría que hacer es generar un autómata con el jflex que reconociera las palabras reservadas y los comentarios para cambiarlos de color. Pero dado que este componente ya incorpora autómatas para reconocer determinados aspectos de diferentes lenguajes, nos bastaría con coger uno de esos autómatas y cambiar la tabla de palabras reservadas y los colores deseados.

11. Apéndice I: Fichero Flex

```
package analizador;

import java_cup.runtime.*;

/* Codigo de usuario */

class Utility {
    public static final String Keywords[] = {"graph","node","edge",
"op", "minimumSpanningTree","shortestPath","union"};
    public static final String errorMsg[] = {
        "Identificador no valido",
        "Error de sintaxis"
    };

    public static boolean isKeyword (String cadena) {
        boolean found = false;
        for (int i = 0; i < Keywords.length && !found; i++) {
                if (cadena.equals(Keywords[i])) found = true;
        }
        return found;
    }

    public static String Keyword (String cadena) {
        String token = "";

        if (cadena.equals("graph")) token = "graph";
        else if (cadena.equals("node")) token = "node";
        else if (cadena.equals("edge")) token = "edge";
        else if (cadena.equals("op")) token = "op";
        else if (cadena.equals("minimumSpanningTree")) token =
"operadorUnario";
        else if (cadena.equals("shortestPath")) token =
"operadorBinario";
        else if (cadena.equals("union")) token = "operadorQuinario";
        else System.out.println (":: ERROR. El lexema no se encuentra
asociado a ningun token.");

        return token;
    }

    public static String errorMsg (int error) {
        return errorMsg[error];
    }

}



%%

/* Seccion de opciones y declaraciones */

%class AnaLex
%cup        /* Compatibilidad con el interfaz CUP */
%public
%line
%column
```

```
%switch
%states COMMENT, COMMENTM

%eof{
    if(zzLexicalState==COMMENTM)
            System.out.println(Utility.errorMsg(ERROR_SYNTAX)+".
Apertura de comentario sin cierre. Linea: "+(yyline+1)+" Columna:
"+(yycolumn+1));
%eof}

%{
public static final int ERROR_MSG_IDENT = 0;
public static final int ERROR_SYNTAX  = 1;

private Symbol symbol (int type) {
    return new Symbol(type,yyline+1,yycolumn+1);
}

private Symbol symbol (int type, Object val) {
    return new Symbol(type,yyline+1,yycolumn+1,val);
}

%}

/* Macros */

LineTerminator = \r | \n | \r\n
WhiteSpace = {LineTerminator} | [\t\f] | " "



%%


<YYINITIAL> {
    "//"        { yybegin(COMMENT); }
    "/*"        { yybegin(COMMENTM); }
    [a-zA-Z][a-zA-Z0-9]*        {
        if (Utility.isKeyword(yytext())) {
            String Keyword = new String();
            Keyword = Utility.Keyword(yytext());
            if (Keyword.equals(yytext())) {
                if (yytext().equals("graph")) return
symbol(sym.graph);
                if (yytext().equals("node")) return symbol(sym.node);
                if (yytext().equals("edge")) return symbol(sym.edge);
                if (yytext().equals("op")) return symbol(sym.op);
            } else {
                if (yytext().equals("minimumSpanningTree")) return
symbol(sym.operadorUnario, yytext());
                if (yytext().equals("shortestPath")) return
symbol(sym.operadorBinario, yytext());
                if (yytext().equals("union")) return
symbol(sym.operadorQuinario, yytext());
            }
        } else {
            return symbol(sym.ident, yytext());
        }
    }
    "{"                 { return symbol(sym.l_bracket); }
    "}"                 { return symbol(sym.r_bracket); }
```

```
    "("                 { return symbol(sym.l_paren); }
    ")"                 { return symbol(sym.r_paren); }
    ";"                 { return symbol(sym.semicolon); }
    ","                 { return symbol(sym.comma); }
    "="                 { return symbol(sym.equal); }
    "-" | "->"          { return symbol(sym.connector, yytext()); }
    0 | [1-9][0-9]*     { return symbol(sym.number, new
Integer(yytext())); }
    "*/"                {
System.out.println(Utility.errorMsg(ERROR_SYNTAX)+". Fin de comentario
sin apertura. Linea: "+(yyline+1)+" Columna: "+(yycolumn+1)); }
    [0-9][a-zA-Z0-9]*  {
System.out.println(Utility.errorMsg(ERROR_MSG_IDENT)+" <"+yytext()+">
Linea: "+(yyline+1)+" Columna: "+(yycolumn+1)); }
    {WhiteSpace}        { }
    .                   { System.out.println("Expresion ilegal
<"+yytext()+"> Linea: "+(yyline+1)+" Columna: "+(yycolumn+1)); }
}



<COMMENT> {
        {LineTerminator}        { yybegin(YYINITIAL); }
        .         { }
}


<COMMENTM> {
        "*/"         { yybegin(YYINITIAL); }
        {LineTerminator}        { }
        .         { }
}
```

12. Apéndice II: Fichero CUP

```
/* Especificaciones de package e imports */
package analizador;

import java_cup.runtime.*;
import dominio.*;
import java.util.ArrayList;
import java.util.Hashtable;
import java.util.Enumeration;

/* Componentes de codigo de usuario */
parser code {:
  public static final int GLOBAL = 0;
  public static final int LOCAL = 1;
  public static final int NODE = 0;
  public static final int EDGE = 1;
  public static final int GRAPH = 2;

  public int ambito = GLOBAL;
  public boolean dirigido = false;
  public boolean no_dirigido = false;
  public boolean valorado = false;
  public boolean no_valorado = false;
  public boolean errorAnalisis = false;

  private ArrayList<Graph> grafos = new ArrayList<Graph>();
  private String mensajes = new String("");

  public ArrayList<Graph> getGrafos () {
    return grafos;
  }

  public void setGrafos (ArrayList<Graph> lg) {
    grafos = lg;
  }

  public String getMensajes() {
    return mensajes;
  }

  private Hashtable<ClaveTS, EntradaTS> tablaSimbolos = new
Hashtable<ClaveTS, EntradaTS>();

  public void borrarTS() {
    tablaSimbolos.clear();
  }

  public boolean existeSimbolo(ClaveTS clave) {
    return tablaSimbolos.containsKey(clave);
  }

  public boolean existeSimbolo(String identificador, int scope) {
    ClaveTS c = new ClaveTS(identificador, scope);
    return tablaSimbolos.containsKey((ClaveTS)c);
  }

  public Hashtable<ClaveTS, EntradaTS> getTabla() {
```

```
    return tablaSimbolos;
  }

  public void insertarSimbolo(String nombre, int scope, Object objeto)
{
    ClaveTS c = new ClaveTS(nombre, scope);
    EntradaTS e = new EntradaTS(objeto, 0);
    tablaSimbolos.put(c, e);
  }

  public void insertarSimbolo(String nombre, int scope, Object objeto,
int timesUsed) {
    ClaveTS c = new ClaveTS(nombre, scope);
    EntradaTS e = new EntradaTS(objeto, timesUsed);
    tablaSimbolos.put(c, e);
  }

  public void insertarSimbolo(ClaveTS c, EntradaTS e) {
    tablaSimbolos.put(c, e);
  }

  public EntradaTS obtenerSimbolo(String nombre, int scope) {
    ClaveTS c = new ClaveTS(nombre, scope);
    return tablaSimbolos.get(c);
  }

  public EntradaTS obtenerSimbolo(ClaveTS clave) {
    return tablaSimbolos.get(clave);
  }

  public int obtenerTipoSimbolo(String nombre) {
    return tablaSimbolos.get(nombre).getTipo();
  }

  public void mostrarError (String mensaje, int fila, int columna) {
    errorAnalisis = true;
    mensajes += fila + ":" + columna + " Semantic error: " + mensaje +
"\n";
  }

  public void mostrarWarning (String mensaje) {
    mensajes += "Warning: " + mensaje + "\n";
  }

  public void mostrarWarning (String mensaje, int fila, int columna) {
    mensajes += fila + ":" + columna + " Warning: " + mensaje + "\n";
  }


  public void resetearAnalisis (){
    Enumeration<ClaveTS> elements;
    ClaveTS clave;
    EntradaTS entrada;
    Edge aristaAux = null;
    Node nodoAux = null;

    dirigido = false;
    no_dirigido = false;
    valorado = false;
    no_valorado = false;
    errorAnalisis = false;
```

```
    elements = tablaSimbolos.keys();
    while (elements.hasMoreElements()) {
      clave = elements.nextElement();
      entrada = tablaSimbolos.get(clave);
      if (clave.getScope() == LOCAL) {
        if (entrada.getTimesUsed() == 0) { //Warning
          if (entrada.getTipo() == EDGE) {
            aristaAux = (Edge)entrada.getVariable();
            mostrarWarning ("Edge " + aristaAux.getName() + " not
used. Assuming it does not exist.");
          }
          else { // NODE
            nodoAux = (Node)entrada.getVariable();
            mostrarWarning ("Node " + nodoAux.getName() + " not used.
Assuming it does not exist.");
          }
        }
        tablaSimbolos.remove(clave);
      }
      else { // GLOBAL
        // reseteamos las veces usadas
        entrada.setTimesUsed(0);
        if (entrada.getTipo() == EDGE) {
          Edge e = (Edge)entrada.getVariable();
          e.setFrom(null);
          e.setTo(null);
          e.setWeight(0);
        }
      }
    }
  }

  public void report_error (String message, Object info) {
    StringBuffer m = new StringBuffer("");

    if (info instanceof java_cup.runtime.Symbol) {
      java_cup.runtime.Symbol s = ((java_cup.runtime.Symbol) info);
      if (s.left >= 0) {
        m.append(""+(s.left));
        if (s.right >= 0)
          m.append(":"+(s.right)+" ");
      }
    }
    m.append(message);
    System.err.println(m);
    mensajes += m + "\n";
  }

  public void report_fatal_error(String message, Object info) {
    report_error(message,info);
    //System.exit(1);
  }
:};

/* Lista de simbolos de la gramatica (terminales y no terminales) */
terminal graph, node, edge, op, operadorUnario, operadorBinario,
operadorQuinario;
terminal l_bracket, r_bracket, l_paren, r_paren;
terminal semicolon, comma, equal;
terminal String ident, connector;
terminal Integer number;
```

```
non terminal SYNTAX, DECL, DECL_NODES, DECL_EDGES, IDENTS_NODE,
IDENTS;
non terminal IDENTS_SIMPLE, IDENTS_WEIGHTED, ARCS, ARC, OPS_GEN, OP5;
non terminal ArrayList<Operation> OPS, BODY;
non terminal Operation OP, OP1, OP2;
non terminal Graph GRAPH;
non terminal ArrayList<Graph> GRAPHS_SET;

/* Declaraciones de precedencia */

/* Especificacion de la gramatica */
SYNTAX    ::=    {: parser.ambito = parser.LOCAL; :}
GRAPHS_SET:listaGrafos
  {:
    parser.resetearAnalisis();
    parser.setGrafos((ArrayList<Graph>)listaGrafos);
  :}
|   DECL {: parser.ambito = parser.LOCAL; :} GRAPHS_SET:listaGrafos
  {:
    parser.resetearAnalisis();
    parser.setGrafos((ArrayList<Graph>)listaGrafos);
  :}
;
GRAPHS_SET   ::= GRAPH:grafo
  {:
    ArrayList<Graph> listaGrafos = new ArrayList<Graph>();
    if ((Graph)grafo != null) listaGrafos.add(0, (Graph)grafo);
    RESULT = (ArrayList<Graph>)listaGrafos;
  :}
|   GRAPH:grafo {: parser.resetearAnalisis(); :}
GRAPHS_SET:listaGrafos
  {:
    if ((Graph)grafo != null) ((ArrayList<Graph>)listaGrafos).add(0,
(Graph)grafo);
    RESULT = (ArrayList<Graph>)listaGrafos;
  :}
;
GRAPH     ::= graph ident:identificador_grafo l_bracket
BODY:lista_operaciones r_bracket
  {:
    Graph gr = null;
    if (!parser.existeSimbolo(identificador_grafo, parser.GLOBAL)) {
      if (!parser.errorAnalisis) {
        //Crear el grafo con los simbolos de ambito 0 y 1 usados mas
de una vez (con clone)
        ArrayList<Node> nodes = new ArrayList<Node>();
        ArrayList<Edge> edges = new ArrayList<Edge>();
        Enumeration<ClaveTS> elements;
        ClaveTS clave;
        EntradaTS entrada;
        Edge aristaAux = null;
        Node nodoAux = null;

        elements = parser.getTabla().keys();
        while (elements.hasMoreElements()) {
          clave = elements.nextElement();
          entrada = parser.getTabla().get(clave);
          if (entrada.getTimesUsed() > 0) {
            if (entrada.getTipo() == parser.EDGE) {
              aristaAux = (Edge)entrada.getVariable();
              edges.add(0, (Edge)aristaAux.clone());
```

```
            }
            if (entrada.getTipo() == parser.NODE) {
              nodoAux = (Node)entrada.getVariable();
              nodes.add(0, (Node)nodoAux.clone());
            }
          }
        }
        //Añadirle la lista de operaciones que ha subido de body,
comprobando antes que existen.
        gr = new Graph((String)identificador_grafo, parser.valorado,
parser.dirigido, nodes, edges,
(ArrayList<Operation>)lista_operaciones);
        //Añadir el grafo a la tabla de simbolos con ambito global
        parser.insertarSimbolo(identificador_grafo, parser.GLOBAL,
gr);
      }
      else {
        parser.mostrarError ("Graph " + identificador_grafo + " not
generated.", identificador_grafoleft, identificador_graforight);
      }
    }
    else {
      parser.mostrarError(identificador_grafo + " is already
defined.", identificador_grafoleft, identificador_graforight);
    }
    RESULT = (Graph)gr;
  :}
| error
  {: parser.report_error("Syntax error in graph definition.",null); :}
;
BODY     ::= DECL ARCS OPS:lista_operaciones
  {:
    RESULT = (ArrayList<Operation>)lista_operaciones;
  :}
|   OPS_GEN   OPS:lista_operaciones
  {:
    RESULT = (ArrayList<Operation>)lista_operaciones;
  :}
| error
  {: parser.report_error("Syntax error in declaration zone.",null); :}
;
DECL     ::= DECL_NODES
|   DECL_EDGES
;
DECL_NODES   ::= node IDENTS_NODE semicolon
|   node IDENTS_NODE semicolon DECL
;
IDENTS_NODE  ::= ident:identificador
  {:
    boolean existe = false;
    if (parser.existeSimbolo(identificador, parser.ambito)) {
      existe = true;
      parser.mostrarError(identificador + " is already defined.",
identificadorleft, identificadorright);
    }
    if (!existe) {
      Node nodo = new Node(identificador);
      parser.insertarSimbolo(identificador, parser.ambito, nodo);
    }
  :}
|   ident:identificador comma IDENTS_NODE
```

```
  {:
    boolean existe = false;
    if (parser.existeSimbolo(identificador, parser.ambito)) {
      existe = true;
      parser.mostrarError(identificador + " is already defined.",
identificadorleft, identificadorright);
    }
    if (!existe) {
      Node nodo = new Node(identificador);
      parser.insertarSimbolo(identificador, parser.ambito, nodo);
    }
  :}
;
DECL_EDGES   ::= edge IDENTS semicolon
|   edge IDENTS semicolon DECL
;
IDENTS    ::= IDENTS_SIMPLE
|   IDENTS_WEIGHTED
;
IDENTS_SIMPLE  ::= ident:identificador
  {:
    boolean existe = false;
    if (parser.existeSimbolo(identificador, parser.ambito)) {
      existe = true;
      parser.mostrarError(identificador + " is already defined.",
identificadorleft, identificadorright);
    }
    if (!existe) {
      parser.no_valorado = true;
      if (parser.valorado) parser.mostrarWarning ("Edge " +
identificador + " is not explicitally weighted. Assuming weight 0.",
identificadorleft, identificadorright);
      Edge edge = new Edge(identificador, 0);
      parser.insertarSimbolo(identificador, parser.ambito, edge);
    }
  :}
|   ident:identificador comma IDENTS
  {:
    boolean existe = false;
    if (parser.existeSimbolo(identificador, parser.ambito)) {
      existe = true;
      parser.mostrarError(identificador + " is already defined.",
identificadorleft, identificadorright);
    }
    if (!existe) {
      parser.no_valorado = true;
      if (parser.valorado) parser.mostrarWarning ("Edge " +
identificador + " is not explicitally weighted. Assuming weight 0.",
identificadorleft, identificadorright);
      Edge edge = new Edge(identificador, 0);
      parser.insertarSimbolo(identificador, parser.ambito, edge);
    }
  :}
;
IDENTS_WEIGHTED  ::= ident:identificador l_paren number:peso r_paren
  {:
    boolean existe = false;
    if (parser.existeSimbolo(identificador, parser.ambito)) {
      existe = true;
      parser.mostrarError(identificador + " is already defined.",
identificadorleft, identificadorright);
```

```
    }
    if (!existe) {
      if (peso.intValue() >= 0 && peso.intValue() < Integer.MAX_VALUE)
{
        parser.valorado = true;
        Edge edge = new Edge(identificador, peso);
        parser.insertarSimbolo(identificador, parser.ambito, edge);
      }
      else {
        parser.mostrarError ("Invalid weight for edge " +
identificador, pesoleft, pesoright);
      }
    }
  :}
|   ident:identificador l_paren number:peso r_paren comma IDENTS
  {:
    boolean existe = false;
    if (parser.existeSimbolo(identificador, parser.ambito)) {
      existe = true;
      parser.mostrarError(identificador + " is already defined.",
identificadorleft, identificadorright);
    }
    if (!existe) {
      if (peso.intValue() >= 0 && peso.intValue() < Integer.MAX_VALUE)
{
        parser.valorado = true;
        Edge edge = new Edge(identificador, peso);
        parser.insertarSimbolo(identificador, parser.ambito, edge);
      }
      else {
        parser.mostrarError ("Invalid weight for edge " +
identificador, pesoleft, pesoright);
      }
    }
  :}
;
ARCS     ::= ARC
|   ARC ARCS
;
ARC     ::= ident:identificador_arista equal
ident:identificador_nodo_origen connector:conector
ident:identificador_nodo_destino semicolon
    {:
      EntradaTS entradaSource = null;
      EntradaTS entradaTarget = null;
      EntradaTS entradaEdge = null;
      Node source = null;
      Node target = null;
      Edge edge = null;
      // Comprobamos que existe el nodo origen
      if (parser.existeSimbolo(identificador_nodo_origen,
parser.LOCAL) || parser.existeSimbolo(identificador_nodo_origen,
parser.GLOBAL)) {
        if (parser.existeSimbolo(identificador_nodo_origen,
parser.LOCAL)) {
          entradaSource =
parser.obtenerSimbolo(identificador_nodo_origen, parser.LOCAL);
          if (entradaSource.getTipo() == parser.NODE) {
            source = (Node)entradaSource.getVariable();
          }
          else parser.mostrarError ("Source node " +
```

```
identificador_nodo_origen + " is not a node.",
identificador_nodo_origenleft, identificador_nodo_origenright);
        }
        else { //Global
          entradaSource =
parser.obtenerSimbolo(identificador_nodo_origen, parser.GLOBAL);
          if (entradaSource.getTipo() == parser.NODE)
            source = (Node)entradaSource.getVariable();
          else parser.mostrarError ("Source node " +
identificador_nodo_origen + " is not a node.",
identificador_nodo_origenleft, identificador_nodo_origenright);
        }
        // Comprobamos que existe el nodo destino
        if (parser.existeSimbolo(identificador_nodo_destino,
parser.LOCAL) || parser.existeSimbolo(identificador_nodo_destino,
parser.GLOBAL)) {
          if (parser.existeSimbolo(identificador_nodo_destino,
parser.LOCAL)) {
            entradaTarget =
parser.obtenerSimbolo(identificador_nodo_destino, parser.LOCAL);
            if (entradaTarget.getTipo() == parser.NODE) {
              target = (Node)entradaTarget.getVariable();
            }
            else parser.mostrarError ("Target node " +
identificador_nodo_destino + " is not a node.",
identificador_nodo_destinoleft, identificador_nodo_destinoright);
          }
          else { //Global
            entradaTarget =
parser.obtenerSimbolo(identificador_nodo_destino, parser.GLOBAL);
            if (entradaTarget.getTipo() == parser.NODE)
              target = (Node)entradaTarget.getVariable();
            else parser.mostrarError ("Target node " +
identificador_nodo_destino + " is not a node.",
identificador_nodo_destinoleft, identificador_nodo_destinoright);
          }
          // Comprobamos que existe la arista
          if (parser.existeSimbolo(identificador_arista, parser.LOCAL)
|| parser.existeSimbolo(identificador_arista, parser.GLOBAL)) {
            if (parser.existeSimbolo(identificador_arista,
parser.LOCAL)) {
              entradaEdge =
parser.obtenerSimbolo(identificador_arista, parser.LOCAL);
              if (entradaEdge.getTipo() == parser.EDGE) {
                edge = (Edge)entradaEdge.getVariable();
              }
              else parser.mostrarError ("Edge " + identificador_arista
+ " is not an edge.", identificador_aristaleft,
identificador_aristaright);
            }
            else { //Global
              entradaEdge =
parser.obtenerSimbolo(identificador_arista, parser.GLOBAL);
              if (entradaEdge.getTipo() == parser.EDGE)
                edge = (Edge)entradaEdge.getVariable();
              else parser.mostrarError ("Edge " + identificador_arista
+ " is not an edge.", identificador_aristaleft,
identificador_aristaright);
            }
            edge.setFrom(source);
            edge.setTo(target);
```

```
            entradaSource.setTimesUsed(entradaSource.getTimesUsed() +
1);
            entradaTarget.setTimesUsed(entradaTarget.getTimesUsed() +
1);
            entradaEdge.setTimesUsed(entradaEdge.getTimesUsed() + 1);
            if (conector.equals("-") && !parser.dirigido)
parser.no_dirigido = true;
            else {
              if (conector.equals("->") && !parser.no_dirigido)
parser.dirigido = true;
              else {
                parser.mostrarError ("All edges must be directed or
not directed.", conectorleft, conectorright);
              }
            }
          }
          else {
            parser.mostrarError ("Edge " + identificador_arista + "
has not been declared.", identificador_aristaleft,
identificador_aristaright);
          }
        }
        else {
          parser.mostrarError ("Target node " +
identificador_nodo_destino + " has not been declared.",
identificador_nodo_destinoleft, identificador_nodo_destinoright);
        }
      }
      else {
        parser.mostrarError ("Source node " +
identificador_nodo_origen + " has not been declared.",
identificador_nodo_origenleft, identificador_nodo_origenright);
      }
    :}
| error
  {: parser.report_error("Syntax error in connection edge
zone.",null); :}
;
OPS_GEN   ::= OP5 semicolon
;
OPS     ::= op OP:operacion
  {:
    ArrayList<Operation> lista_operaciones = new
ArrayList<Operation>();
    if (operacion != null) {
      lista_operaciones.add(0, (Operation)operacion);
    }
    RESULT = (ArrayList<Operation>)lista_operaciones;
  :}
|   OPS:lista_operaciones op OP:operacion
  {:
    if (operacion != null) {
      ((ArrayList<Operation>)lista_operaciones).add(0,
(Operation)operacion);
    }
    RESULT = (ArrayList<Operation>)lista_operaciones;
  :}
;
OP     ::= OP1:operacion semicolon
  {:
    RESULT = (Operation)operacion;
```

```
  :}
|   OP2:operacion semicolon
  {:
    RESULT = (Operation)operacion;
  :}
| error
  {: parser.report_error("Syntax error in operation zone.",null); :}
;
OP1     ::= operadorUnario:identificador_operacion   l_paren
ident:identificador_nodo                          r_paren
  {:
    EntradaTS entrada = null;
    Node source = null;
    Operation o = null;
    if (parser.dirigido &&
identificador_operacion.equals("minimumSpanningTree")) {
      parser.mostrarWarning("Operation " + identificador_operacion + "
can only be applied to undirected graphs.",
identificador_operacionleft, identificador_operacionright);
      parser.mostrarWarning("Operation " + identificador_operacion + "
not executed.", identificador_operacionleft,
identificador_operacionright);
    }
    else {
      if (parser.existeSimbolo(identificador_nodo, parser.LOCAL) ||
parser.existeSimbolo(identificador_nodo, parser.GLOBAL)) {
        if (parser.existeSimbolo(identificador_nodo, parser.LOCAL)) {
          entrada = parser.obtenerSimbolo(identificador_nodo,
parser.LOCAL);
          if (entrada.getTipo() == parser.NODE &&
entrada.getTimesUsed() > 0) {
            source = (Node)entrada.getVariable();
          }
          else parser.mostrarError("Param " + identificador_nodo + "
must be a connected node.", identificador_nodoleft,
identificador_nodoright);
        }
        else { //Global
          entrada = parser.obtenerSimbolo(identificador_nodo,
parser.GLOBAL);
          if (entrada.getTipo() == parser.NODE &&
entrada.getTimesUsed() > 0)
            source = (Node)entrada.getVariable();
          else parser.mostrarError("Param " + identificador_nodo + "
must be a connected node.", identificador_nodoleft,
identificador_nodoright);
        }
        ArrayList<Object> lista = new ArrayList<Object>();
        if (source != null) lista.add(source);
        o = new Operation((String)identificador_operacion, lista);
      }
      else {
        parser.mostrarError("Node " + identificador_nodo + " has not
been declared.", identificador_nodoleft, identificador_nodoright);
      }
    }
    RESULT = (Operation)o;
  :}
;
OP2     ::= operadorBinario:identificador_operacion  l_paren
ident:identificador_nodo_origen comma ident:identificador_nodo_destino
```

```
r_paren
  {:
    EntradaTS entrada = null;
    Node source = null;
    Node target = null;
    Operation o = null;
    if (!parser.dirigido &&
identificador_operacion.equals("shortestPath")) {
      parser.mostrarWarning("Operation " + identificador_operacion + "
can only be applied to directed graphs.", identificador_operacionleft,
identificador_operacionright);
      parser.mostrarWarning("Operation " + identificador_operacion + "
not executed.", identificador_operacionleft,
identificador_operacionright);
    }
    else {
      // Comprobamos el nodo origen
      if (parser.existeSimbolo(identificador_nodo_origen,
parser.LOCAL) || parser.existeSimbolo(identificador_nodo_origen,
parser.GLOBAL)) {
        if (parser.existeSimbolo(identificador_nodo_origen,
parser.LOCAL)) {
          entrada = parser.obtenerSimbolo(identificador_nodo_origen,
parser.LOCAL);
          if (entrada.getTipo() == parser.NODE &&
entrada.getTimesUsed() > 0) {
            source = (Node)entrada.getVariable();
          }
          else parser.mostrarError("First param " +
identificador_nodo_origen + " must be a connected node.",
identificador_nodo_origenleft, identificador_nodo_origenright);
        }
        else { //Global
          entrada = parser.obtenerSimbolo(identificador_nodo_origen,
parser.GLOBAL);
          if (entrada.getTipo() == parser.NODE &&
entrada.getTimesUsed() > 0)
            source = (Node)entrada.getVariable();
          else parser.mostrarError("First param " +
identificador_nodo_origen + " must be a connected node.",
identificador_nodo_origenleft, identificador_nodo_origenright);
        }
        // Comprobamos el nodo destino
        if (parser.existeSimbolo(identificador_nodo_destino,
parser.LOCAL) || parser.existeSimbolo(identificador_nodo_destino,
parser.GLOBAL)) {
          if (parser.existeSimbolo(identificador_nodo_destino,
parser.LOCAL)) {
            entrada =
parser.obtenerSimbolo(identificador_nodo_destino, parser.LOCAL);
            if (entrada.getTipo() == parser.NODE &&
entrada.getTimesUsed() > 0) {
              target = (Node)entrada.getVariable();
            }
            else parser.mostrarError("Second param " +
identificador_nodo_destino + " must be a connected node.",
identificador_nodo_destinoleft, identificador_nodo_destinoright);
          }
          else { //Global
            entrada =
parser.obtenerSimbolo(identificador_nodo_destino, parser.GLOBAL);
```

```
            if (entrada.getTipo() == parser.NODE &&
entrada.getTimesUsed() > 0)
              target = (Node)entrada.getVariable();
            else parser.mostrarError("Second param " +
identificador_nodo_destino + " must be a connected node.",
identificador_nodo_destinoleft, identificador_nodo_destinoright);
          }
          ArrayList<Object> lista = new ArrayList<Object>();
          if (source != null && target != null) {
            lista.add(source);
            lista.add(target);
          }
          o = new Operation((String)identificador_operacion, lista);
        }
        else {
          parser.mostrarError ("Node " + identificador_nodo_destino +
" has not been declared.", identificador_nodo_destinoleft,
identificador_nodo_destinoright);
        }
      }
      else {
        parser.mostrarError ("Node " + identificador_nodo_origen + "
has not been declared.", identificador_nodo_origenleft,
identificador_nodo_origenright);
      }
    }
    RESULT = (Operation)o;
  :}
;
OP5     ::= operadorQuinario:identificador_operacion l_paren
ident:identificador_grafo_origen comma
ident:identificador_grafo_destino comma
ident:identificador_nodo_origen comma ident:identificador_nodo_destino
comma ident:identificador_arista r_paren
  {:
    EntradaTS entrada = null;
    Graph sourceGraph = null;
    Graph targetGraph = null;
    Node source = null;
    Node target = null;
    Edge edge = null;
    Graph graphAux = null;
    Node nodeAux = null;
    Edge edgeAux = null;
    if (parser.existeSimbolo(identificador_grafo_origen,
parser.GLOBAL)) {
      entrada = parser.obtenerSimbolo(identificador_grafo_origen,
parser.GLOBAL);
      if (entrada.getTipo() == parser.GRAPH)
        sourceGraph = (Graph)entrada.getVariable();
      else parser.mostrarError("First param " +
identificador_grafo_origen + " must be a graph.",
identificador_grafo_origenleft, identificador_grafo_origenright);
      // Comprobamos que existe el grafo 2
      if (parser.existeSimbolo(identificador_grafo_destino,
parser.GLOBAL)) {
        entrada = parser.obtenerSimbolo(identificador_grafo_destino,
parser.GLOBAL);
        if (entrada.getTipo() == parser.GRAPH) {
          targetGraph = (Graph)entrada.getVariable();
          if ((sourceGraph.isDirected() && targetGraph.isDirected())
```

```
|| (!sourceGraph.isDirected() && !targetGraph.isDirected())) {
            if ((sourceGraph.isWeighted() &&
!targetGraph.isWeighted()) || (!sourceGraph.isWeighted() &&
targetGraph.isWeighted()))
              parser.mostrarWarning("Both " +
identificador_grafo_origen + " and " + identificador_grafo_destino + "
are not weighted.", identificador_grafo_origenleft,
identificador_grafo_destinoright);
            // Comprobamos que existe el nodo origen en sourceGraph,
si existe es porque el grafo es conexo
            if (sourceGraph.hasNode(identificador_nodo_origen)) {
              source =
(Node)sourceGraph.getNode(identificador_nodo_origen);
              // Comprobamos que existe el nodo destino en targetGraph
              if (targetGraph.hasNode(identificador_nodo_destino)) {
                target =
(Node)targetGraph.getNode(identificador_nodo_destino);
                // Comprobamos que existe la arista globalmente
                if (parser.existeSimbolo(identificador_arista,
parser.GLOBAL)) {
                  entrada =
parser.obtenerSimbolo(identificador_arista, parser.GLOBAL);
                  if (entrada.getTipo() == parser.EDGE) {
                    edge = (Edge)entrada.getVariable();
                    // como se cumplen todas las condiciones...
                    graphAux = Graph.Union(sourceGraph, targetGraph,
source, target, edge);
                    for (Node n : graphAux.getNodes()) {
                      nodeAux = (Node)n;
                      parser.insertarSimbolo(nodeAux.getName(),
parser.LOCAL, nodeAux, 1);
                    }
                    for (Edge e : graphAux.getEdges()) {
                      edgeAux = (Edge)e;
                      parser.insertarSimbolo(edgeAux.getName(),
parser.LOCAL, edgeAux, 1);
                    }
                    parser.valorado = graphAux.isWeighted();
                    parser.dirigido = graphAux.isDirected();
                  }
                  else parser.mostrarError("Fifth param " +
identificador_arista + " must be an edge.", identificador_aristaleft,
identificador_aristaright);
                }
                else {
                  parser.mostrarError("Edge " + identificador_arista +
" must be globally declared.", identificador_aristaleft,
identificador_aristaright);
                }
              }
              else {
                parser.mostrarError("Fourth param " +
identificador_nodo_destino + " must be an existing node in " +
identificador_grafo_destino + ".", identificador_nodo_destinoleft,
identificador_nodo_destinoright);
              }
            }
            else {
              parser.mostrarError("Third param " +
identificador_nodo_origen + " must be an existing node in " +
identificador_grafo_origen + ".", identificador_nodo_origenleft,
```

```
identificador_nodo_origenright);
            }
          }
          else {
            parser.mostrarError(identificador_grafo_origen + " and " +
identificador_grafo_destino + " must be both directed graphs or
undirected graphs.", identificador_grafo_origenleft,
identificador_grafo_destinoright);
          }
        }
        else {
          parser.mostrarError("Second param " +
identificador_grafo_destino + " must be a graph.",
identificador_grafo_destinoleft, identificador_grafo_destinoright);
        }
      }
      else {
        parser.mostrarError("Second param " +
identificador_grafo_destino + " has not been defined.",
identificador_grafo_destinoleft, identificador_grafo_destinoright);
      }
    }
    else {
      parser.mostrarError("First param " + identificador_grafo_origen
+ " has not been defined.", identificador_grafo_origenleft,
identificador_grafo_origenright);
    }
  :}
;
```

13. Referencias

1: Algowiki
http://algowiki.net/wiki/index.php/

2: GraphViz
http://www.graphviz.org

3: Java Sun API
http://java.sun.com/javase/6/docs/api/

4: CUP LALR Parser Generator
http://www2.cs.tum.edu/projects/cup/